PCT

WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : G07F

A2

(11) International Publication Number: **WO 97/50063**

(43) International Publication Date: 31 December 1997 (31.12.97)

(21) International Application Number: PCT/EP97/03355

(22) International Filing Date: 26 June 1997 (26.06.97)

(30) Priority Data:
9613450.7 27 June 1996 (27.06.96) GB

(71) Applicant *(for all designated States except US)*: EUROPAY INTERNATIONAL N.V. [BE/BE]; Chaussée de Tervuren 198A, B-1410 Waterloo (BE).

(72) Inventors; and
(75) Inventors/Applicants *(for US only)*: HEYNS, Guido [BE/BE]; Eekhoornlaan 23B, B-3210 Linden (BE). JOHANNES, Peter [BE/BE]; Edelzangerslaan 56/11, B-3010 Kessel-Lo (BE).

(74) Agents: BIRD, William, E. et al.; Bird Goen & Co., Termerestraat 1, B-3020 Winksele (BE).

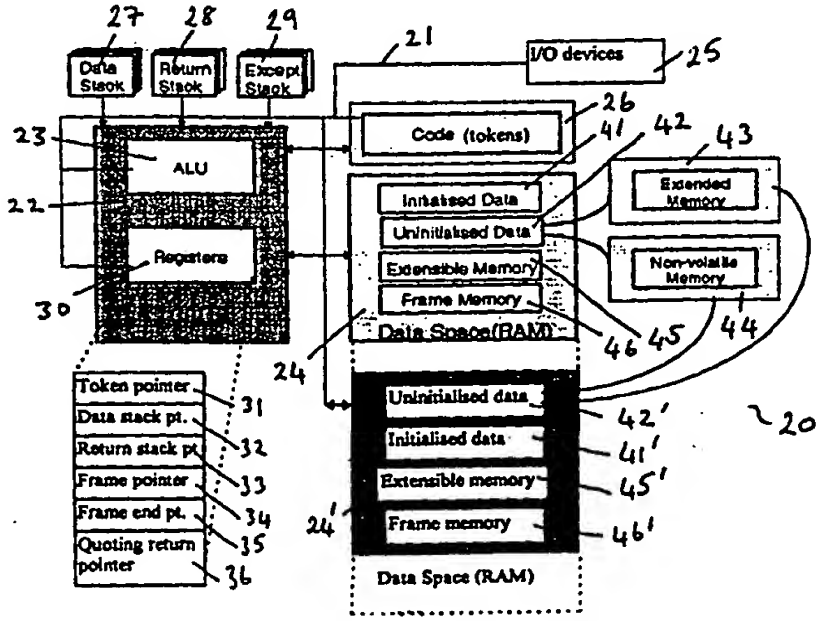
(81) Designated States: AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CU, CZ, DE, DK, EE, ES, FI, GB, GE, GH, HU, IL, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, TJ, TM, TR, TT, UA, UG, US, UZ, VN, YU, ZW, ARIPO patent (GH, KE, LS, MW, SD, SZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, ML, MR, NE, SN, TD, TG).

Published
Without international search report and to be republished upon receipt of that report.

(54) Title: PORTABLE, SECURE TRANSACTION SYSTEM FOR PROGRAMMABLE, INTELLIGENT DEVICES

(57) Abstract

The present invention provides a transaction management system for executing transactions between a first device (1) and a second device, said first and second devices being adapted for communication with each other and at least one of said first and second devices being an integrated circuit card, said system comprising: at least one input/output device (25); a portable virtual machine (20) for interpreting a computer program on said first device, said virtual machine comprising a virtual microprocessor and a driver for said at least one input/output device (25); and execution means responsive to said interpreted program for executing said program. The general linking technical concept behind the present invention is portability combined with security of data and run-time guarantees in a transaction system which are independent of the target implementation provided compile time checks are passed successfully. This concept is achieved by: using a virtual machine as an interpreter, including a driver for the I/O devices in the virtual machine so that application programs have a common interface with I/O devices and are therefore portable across widely differing environments, allocating and deallocating memory and including an indication of the amount of memory in the application program which means that the program will only run successfully or it will not run at all and security management functions are reduced to a minimum which improves operating speed, and providing a secure way of importing and exporting data in and out of application programs and databases.



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania
AM	Armenia
AT	Austria
AU	Australia
AZ	Azerbaijan
BA	Bosnia and Herzegovina
BB	Barbados
BE	Belgium
BF	Burkina Faso
BG	Bulgaria
BJ	Benin
BR	Brazil
BY	Belarus
CA	Canada
CF	Central African Republic
CG	Congo
CH	Switzerland
CI	Côte d'Ivoire
CM	Cameroon
CN	China
CU	Cuba
CZ	Czech Republic
DE	Germany
DK	Denmark
EE	Estonia
ES	Spain
FI	Finland
FR	France
GA	Gabon
GB	United Kingdom
GE	Georgia
GH	Ghana
GN	Guinea
GR	Greece
HU	Hungary
IE	Ireland
IL	Israel
IS	Iceland
IT	Italy
JP	Japan
KE	Kenya
KG	Kyrgyzstan
KP	Democratic People's Republic of Korea
KR	Republic of Korea
KZ	Kazakstan
LC	Saint Lucia
LI	Liechtenstein
LK	Sri Lanka
LR	Liberia
LS	Lesotho
LT	Lithuania
LU	Luxembourg
LV	Latvia
MC	Monaco
MD	Republic of Moldova
MG	Madagascar
MK	The former Yugoslav Republic of Macedonia
ML	Mali
MN	Mongolia
MR	Mauritania
MW	Malawi
MX	Mexico
NE	Niger
NL	Netherlands
NO	Norway
NZ	New Zealand
PL	Poland
PT	Portugal
RO	Romania
RU	Russian Federation
SD	Sudan
SE	Sweden
SG	Singapore
SI	Slovenia
SK	Slovakia
SN	Senegal
SZ	Swaziland
TD	Chad
TG	Togo
TJ	Tajikistan
TM	Turkmenistan
TR	Turkey
TT	Trinidad and Tobago
UA	Ukraine
UG	Uganda
US	United States of America
UZ	Uzbekistan
VN	Viet Nam
YU	Yugoslavia
ZW	Zimbabwe

Portable, secure transaction system for programmable, intelligent devices

The present invention relates to a system including programmable, intelligent devices such as terminals and integrated circuit cards as well as a method of operation of such cards and terminals including automatic teller machines, personal computers, pay-television units, point of sale terminals, health cards or similar. It is particularly suitable for use in performing financial transactions.

Technical background

Various types of terminals are known for the performance of transactions, e.g. financial transactions which involve transfer or exchange of value or transactions which are of a commercial nature such as transactions with health care cards or for accessing data in general, e.g. the SIM card of a GSM mobile telephone. Terminals such as point of sale (POS) devices, automatic teller machines (ATM) or GSM mobile telephones are known. Actual products range from small, hand-held devices with simple 8-Bit microprocessors such as the Intel 8031/8051 series supplied by Intel Corp., USA or Integrated Circuit Cards (ICC) to 32-bit computers running operating systems such as UNIX™, or Windows NT supplied by Microsoft Corp., USA. Some of these machines interact with a personal user card which may be a magnetic stripe card, smartcard or ICC which stores specific user identification and validation information necessary before communication can be initiated between the user and the terminal. The user places the card in a card reader associated with the terminal, a terminal resident program in the terminal is executed and examines the card, checking the user information for its validity and if necessary prompting for a password or private number such as a PIN (personal identification number). After validation, the program normally allows the user to select the desired services to be performed, e.g. withdrawal of cash, examination of current balance. The terminal may be stand-alone or be connected to a larger computer either locally or via a telecommunications network. Such terminals are often available 24 hours a day and must run with the minimum of maintenance and with a high level of security.

Such terminals represent a significant investment in hardware and are not normally replaced at frequent intervals. Updating of the software and programs running on such terminals becomes necessary as new services are offered and must be carried out securely. Generally, the terminal operating organizations such as banks require each update to be certified. Such updating may be by hand or remotely via a private or public

communications network as is known from US Patent No. 5,434,999. Such known schemes require the type and model of the terminal to be known for new developments as the software for each terminal must be created specifically for that type of terminal and are therefore cost intensive. Further, to be able to offer services from all possible organizations offering similar services, e.g. all banks or credit institutes, the terminal must be able to process all the programs of all the organizations. Due to the high mobility of both private and business people, it is advantageous that all services which are offered in one country should be available on each terminal. This would result in unnecessarily large processing capacity and memory size for each terminal. Further, each of these programs must be updated as required. One solution could be using a small workstation for each terminal possibly connected to a telecommunications system. Such a system would be capable of off-line processing and could switch to on-line processing of unusual transactions or for automatic updating of resident programs. Workstations would be required, for instance, to carry out the complex validation and encryption schemes necessary to maintain security on a system open to attack via public telephone networks. With increasing size and complexity, the problem of maintaining security would also increase.

Even with such a system there may be problems with version control. Not all users of the services of the same organization may have cards suitable for the latest version of a service. This can occur when multi-national organizations introduce or update services at different times in different countries. It has been proposed in WO 96/18979 to update terminals merely for the session involved from the personal ICC of the user. Program instructions representing subroutines are stored on the card and can be exported to the terminal where they are interpreted. The use of an interpreter in the terminal allows the same card to be used with any terminal containing the interpreter and therefore makes the transaction independent of the host processor on the terminal. However, no method of security control is described to eliminate potentially dangerous subroutines.

Terminals of the type described above also have a processor including some form of memory, usually some random access memory (RAM) for running programs, some read only memory (ROM) for storing data which only has to be read which can include the program for the operating system of the terminal and non-volatile read/write memory for the storing of general data which may change. The user personal data should be kept private and therefore there should be no possibility of access by one user to the data of others, either accidentally or intentionally by a malicious person. Further, the various writable memories of the terminal should not become defragmented with time.

Defragmentation of memory can result in blocks of contiguous memory being reduced in size so that certain programs cannot run. To avoid this problem some programming languages such as Java™ make use of garbage collection. Garbage collection is a routine which attempts to identify data in memory which is no longer required and deallocates it. It is current informed opinion that garbage collection is a more reliable way of managing memory than having a program explicitly deallocate its own stored data. Explicit memory allocation and deallocation is held by some to be the largest single source of programming errors in common high level programming languages such as C or C++.

Garbage collection has several disadvantages. Firstly, garbage collection is an operating system function rather than an application specific one. Therefore, garbage collection does not guarantee that the data of each application is deallocated at the end of the application, but rather such data may be present for some time until the lack of access triggers the garbage collection. A more secure method of elimination of the possibility of addressing private user data is required in financial transactions. Secondly, it increases the size of memory space required for the operating system. On ICC's and some terminals memory can be limited and use of garbage collection can be a serious disadvantage. As explained above, terminals are replaced very infrequently so that a wide variety of terminals including different processor capabilities and memory sizes are normally operated in the system concurrently. Older terminals are often severely limited in their capabilities. Although the oldest types can be replaced, the demand for more refined and complex services means that older terminals will probably never be replaced so often that some of them do not lag behind in their capabilities. Hence, the requirement for compact operating systems which can work on a wide variety of processor types will probably remain as a requirement. Finally, garbage collection does not free memory as soon as it would be freed using explicit deallocation. This can also increase the amount of memory required as memory is tied up when it could be released..

One secure method of run-time memory management is described in US Patent No. 5,434,999. For instance, in accordance with this known method, an interpreter in the terminal performs systematic checking of any instruction manipulating a memory address in order to verify if the region of the memory to which access is requested is authorized. This system has the disadvantage that every instruction must be checked in this way which slows down processing considerably. Program run-time checking is costly in performance.

There is a need for a system providing programmable terminals which allows the application programmer to generate software that is portable and neutral across

heterogeneous terminals, i.e. independent of the processor used in the terminal, and does not need to be type-approved for each type or make of terminal. The terminal resident operating system and the application programs are preferably compact, execute rapidly, and meet security requirements. Further, it is preferable if the application programs can be updated easily, at least that each user can obtain the services expected independent of the geographical location of the terminal.

An object of the present invention is to provide a secure transaction management system for transactions and a method of operating such a system.

It a further object of the present invention to provide secure terminals and ICC's for transactions and methods of operating such devices.

It is still a further object of the invention to provide a device usable in a transaction which can be implemented on small hand-held devices such as an ICC.

It is yet another object of the present invention to provide transaction system in which the terminals or ICC's can be updated using the terminals or ICC's as sources of the update information.

It is a further object of the present invention to provide a transaction management system and a method of operating the system which provides high security with good speed of operation.

Summary of the invention

The present invention relates to a transaction management system for executing transactions between a first device and a second device, said first and second devices being adapted for communication with each other and at least one of said first and second devices being an integrated circuit card, said system comprising: at least one input/output device;

a portable virtual machine for interpreting a computer program on said first device, said virtual machine comprising a virtual microprocessor and a driver for said at least one input/output device; and execution means responsive to said interpreted program for executing said program.

It is preferred if the portable virtual machine is a stack machine as this brings operating speed and compactness.

The present invention also provides a terminal comprising a first device for carrying out a transaction with a second device, at least one of said first and second devices being an integrated circuit card, comprising: a portable virtual machine interpreting a computer

program on said first device, said portable virtual machine comprising a virtual microprocessor and a driver for at least one input/output device, and execution means responsive to said interpreted program for executing said program.

The present invention also provides a self-contained portable intelligent card including a first device for carrying out a transaction with a second device, said intelligent card comprising: a portable virtual machine comprising a virtual microprocessor and a driver for at least one input/output device.

The present invention also provides a transaction management system comprising: a first device and a second device, said first and second devices being adapted for communication with each other, at least one of said first and second devices being an integrated circuit card; said second device including means for providing at least one program instruction capable of at least modifying the execution time behavior of a computer program on said first device; said first device including a virtual machine, said virtual machine comprising means for loading and interpreting said computer program, said means for loading and interpreting being further adapted to load and interpret said at least one program instruction dependent upon a pre-defined security condition after said means for loading and interpreting has loaded said computer program and while said computer program is running; and execution means for executing said loaded and interpreted computer program with said modified behavior in response to said loaded and interpreted program instruction.

Further the present invention provides a. terminal comprising a first device for carrying out a transaction with a second device and at least one of said first and second devices being an integrated circuit card, said second device including means for providing at least one program instruction capable of at least modifying the execution time behavior of a computer program on said first device; said terminal comprising: said first device including a virtual machine, said virtual machine comprising means for loading and interpreting said computer program, said means for loading and interpreting being further adapted to load and interpret said at least one program instruction dependent upon a pre-defined security condition after said means for loading and interpreting has loaded said computer program and while said computer program is running; and execution means for executing said loaded and interpreted computer program with said modified behavior in response to said loaded and interpreted program instruction.

The present invention provides a self-contained portable intelligent card including a first device for carrying out a transaction with a second device, said second device

including means for providing at least one program instruction capable of at least modifying the execution time behavior of a computer program on said first device, said intelligent card comprising: said first device including a virtual machine, said virtual machine comprising means for loading and interpreting said computer program, said means for loading and interpreting being further adapted to load and interpret said at least one program instruction dependent upon a pre-defined security condition after said means for loading and interpreting has loaded said computer program and while said computer program is running; and execution means for executing said loaded and interpreted computer program with said modified behavior in response to said loaded and interpreted program instruction.

The present invention also provides a transaction system for executing transactions between a first device and a second device, said system comprising: a virtual machine for interpreting a set of customized byte code tokens applied thereto; said virtual machine including a virtual processing unit and read/writable logical address space; at least one first application program including an indication of the amount of read/writable logical address space needed for its execution, said at least one first application program being written as a stream of tokens selected from said set of tokens and corresponding in-line data; said virtual machine also including: a loader for loading said at least one first application program; and means for allocating a first amount of read/writable logical address space specifically for said at least one first application program in accordance with said indication, said allocated read/writable logical address space having defined and protected boundaries. The first device in accordance with the present invention may be a personal computer connected to the Internet and running a browser, the requirement that each module received by the browser, must contain an indication of its memory requirements, improves the security of the browser and limits the damage that could be done by any virus contained in the imported module

The present invention provides a terminal comprising a first device for executing transactions with a second device, said first device comprising: a virtual machine for interpreting a set of customized byte code tokens applied thereto; said virtual machine including a virtual processing unit and read/writable logical address space; at least one first application program including an indication of the amount of read/writable logical address space needed for its execution and a first exclusive list of at least one function which can be exported to other application programs, said at least one first application program being written as a stream of tokens selected from said set of

tokens and corresponding in-line data; said virtual machine also including:
a loader for loading said at least one first application program; and means for allocating a first amount of read/writable logical address space specifically for said at least one first application program in accordance with said indication, said allocated read/writable logical address space having defined and protected boundaries.

The present invention may also provide a self-contained portable intelligent card including a first device for carrying out a transaction with a second device, said first device comprising: a virtual machine for interpreting a set of customized byte code tokens applied thereto; said virtual machine including a virtual processing unit and read/writable logical address space; at least one first application program including an indication of the amount of read/writable logical address space needed for its execution, said at least one first application program being written as a stream of tokens selected from said set of tokens and corresponding in-line data; said virtual machine also including:
a loader for loading said at least one first application program; and means for allocating a first amount of read/writable logical address space specifically for said at least one first application program in accordance with said indication, said allocated read/writable logical address space having defined and protected boundaries.

The present invention may also provide a transaction system for executing transactions between a first device and a second device, at least one of said first and second devices being an integrated circuit card, said system comprising: a virtual machine for interpreting a set of customized byte code tokens applied thereto; said virtual machine including a virtual processing unit and read/writable logical address space; at least one database including at least one record and at least one computer program for execution by said virtual machine, said computer program being a module written in terms of a stream of said tokens selected from said set and including an indication of the amount of uninitialised read/writable logical address space necessary for execution of said module; a loader for loading said module and for allocating the required amount of uninitialised logical address/space in accordance with said indication; and means for accessing a record in said database, records in said database only being accessible through said module, and said accessing means providing a window onto a current record of said database and copying said record into a portion of said uninitialised read/writable logical address space addressable by said application program.

Further, the present invention may also provide a terminal comprising a first device for executing transactions with a second device, at least one of said first and second

devices being an integrated circuit card, said first device comprising: a virtual machine for interpreting a set of customized byte code tokens applied thereto; said virtual machine including a virtual processing unit and read/writable logical address space; at least one database including at least one record and at least one computer program for execution by said virtual machine, said computer program being a module written in terms of a stream of tokens selected form said set and including an indication of the amount of uninitialised read/writable logical address space necessary for execution of said module; a loader for loading said module and for allocating the required amount of uninitialised logical address/space in accordance with said indication; and means for accessing a record in said database, records in said database only being accessible through said module, and said accessing means providing a window onto a current record of said database and copying said record into a portion of said uninitialised read/writable logical address space addressable by said application program.

The present invention may provide a self-contained portable intelligent card including a first device for carrying out a transaction with a second device, said first device comprising: a virtual machine for interpreting a set of customized byte code tokens applied thereto; said virtual machine including a virtual processing unit and read/writable logical address space; at least one database including at least one record and at least one computer program for execution by said virtual machine, said computer program being a module written in terms of a stream of tokens selected form said set and including an indication of the amount of uninitialised read/writable logical address space necessary for execution of said module; a loader for loading said module and for allocating the required amount of uninitialised logical address/space in accordance with said indication; and means for accessing a record in said database, records in said database only being accessible through said module, said accessing means providing a window onto a current record of said database and copying said record into a portion of said uninitialised read/writable logical address space addressable by said application program.

The present invention also provides a method of carrying out a transaction between a first device and a second device, at least one of said first and second devices being an integrated circuit card; comprising: providing at least one program instruction on said second device capable of at least modifying the execution time behavior of a computer program on said first device; loading and interpreting said computer program, loading and interpreting said at least one program instruction dependent upon a pre-defined security condition while said computer program is running; and executing said

loaded and interpreted computer program with said modified behavior in response to said loaded and interpreted program instruction.

The present invention also provides a method of carrying out a transaction between a first device and a second device, comprising: interpreting at least one application program written as a stream of byte code tokens selected from a set of tokens and corresponding in-line data; loading said at least one application program; allocating a first amount of read/writable logical address space specifically for said at least one application program in accordance with an indication contained within said application program of the amount of read/writable logical address space needed for its execution , and defining and protecting the boundaries of said allocated read/writable logical address space. This method combines the use of an interpreter with allocation and optional explicit deallocation of memory. This provides a mixture of flexibility and portability while providing run-time guarantees after the application program has been thoroughly checked at the compiling stage. It reduces the damage which can be caused by viruses in imported application modules.

The present invention also includes a method of carrying out a transaction system between a first device and a second device, at least one of said first and second devices being an integrated circuit card, comprising: interpreting the tokens in a module written in terms of a stream of said tokens selected from a set of tokens; allocating an amount of uninitialised logical address/space in accordance with an indication in said module of the amount of uninitialised read/writable logical address space necessary for execution of said module; accessing a record in a database by providing a window onto a current record of said database, records in a database only being accessible through said module; and copying said record into a portion of said uninitialised read/writable logical address space addressable by said module.

The present invention also including a method of carrying out a transaction between a first device and a second device, at least one of said first and second devices being an integrated circuit card, comprising: providing a portable virtual machine comprising a virtual microprocessor and a driver for at least one input/output device; interpreting a computer program on said first device using said portable virtual machine; and executing said program in response to said interpreted program.

In accordance with the present invention a secure transaction management system is provided which preferably includes a portable virtual microprocessor. Preferably, each module owns a set of virtual address spaces guaranteed to be distinct from any other

virtual address space. Also the portable virtual microprocessor preferably, protects access to shared resources such as the various stacks or the databases. Preferably, the minimum protection is memory checking the bounds of data space access for reads and writes and absolute prohibition of writes to code space. Further, checking is preferred for underflow and overflow on data and return stacks. Preferably, a module can only access what some other module explicitly exports. Preferably, there is no way unexported data can be accessed (the virtual microprocessor doesn't leak) except through functions provided by a module. Modules can preferably not export data in the usual sense; modules can preferably only export functions. Preferably, logical borders forbid data space leakage. In other words, all the data owned by a module is preferably strictly private. This restriction is preferably enforced both at compile time and at runtime, as modules have separate address spaces, which means that the address of some data within some module is completely meaningless outside its owning module.Preferably, a module can only export a set of *handles* to trigger on or off a particular behavior. Preferably, well behaving modules will run flawlessly, while not so well behaving modules will be aborted by exceptions directly thrown by the portable virtual microprocessor when an illegal operation is attempted

The dependent claims define individual embodiments of the present invention. The present invention, its embodiments and advantages will now be described with reference to the following drawings.

Brief description of the drawings

Fig. 1 is a schematic representation of a terminal in accordance with the present invention.

Fig. 2 is a schematic representation of an ICC in accordance with the present invention.

Fig. 3 is a schematic flow diagram of the process of developing and executing a module in accordance with the present invention.

Fig. 4 is a schematic representation of a portable vertical microprocessor in accordance with the present invention as implemented on a terminal.

Fig. 5 is a schematic representation of the portable vertical microprocessor in accordance with the present invention.

Fig. 6 is a schematic representation of a modules loaded into memory in accordance with the present invention.

Fig. 7 is a schematic representation of the method of obtaining access to a database record in accordance with the present invention.

Fig. 8 is a schematic representation of the plug and socket procedure in accordance with

the present invention.

Fig. 9 is a flow diagram of the module loading procedure of the present invention.

Fig. 10 is a flow diagram of the module executing procedure of the present invention.

Fig. 11 is a flow diagram of the socket plugging procedure of the present invention.

Fig. 12 is a flow diagram of the card module loading procedure in accordance with the present invention.

Appendix giving the token codes and standard exceptions

Description of the illustrative embodiments

The present invention will be described in the following with reference to particular drawings and certain embodiments but the invention is not limited thereto but only by the claims. The drawings are only schematic and are not limiting. The present invention will be described with reference to financial transactions but the invention is not limited thereto. Further, the present invention will be described mainly with reference to a terminal but the present invention also includes providing the portable virtual microprocessor in accordance with the present invention on any suitable device, e.g. a personal computer (PC), an ICC or a combined ICC and interface as described in WO94/10657 which is incorporated herein by reference.

The general linking technical concept behind the present invention is portability combined with security of data and run-time guarantees in a transaction system which are independent of the target implementation provided compile time checks are passed successfully. This concept is achieved by one or more of the following features: using a virtual machine as an interpreter, including a driver for the I/O devices in the virtual machine so that application programs have a common interface with I/O devices and are therefore portable across widely differing environments, including an indication of the amount of memory in the application program and allocating memory in accordance with the indication, explicitly deallocating memory and providing a secure way of importing and exporting data in and out of application programs and/or databases.

Fig. 1 is a schematic representation of a terminal 1 in accordance with the present invention. Typically, the terminal 1 includes a central processor unit (CPU) 2 which is connected with memory 4 and input/output (I/O) devices 6 via a bus 3 for two way communication. I/O devices 6 may be a keyboard for entering data and a screen such as a visual display unit, e.g. a liquid crystal (LCD) or light emitting diode (LED) display, for displaying the progress of the transaction and/or for displaying messages or prompts. One

of the I/O devices 6 may be a card reader 7 with which an ICC 5 may be read when it is introduced into the receiving slot in the reader 7. The actual form of the terminal may vary greatly, e.g. it may be a point of sale (POS) terminal and may include processors from an Intel 8051 to a Pentium™. Further, it is not necessary that the terminal 1 is all situated at one location, the various parts of the terminal such as the card reader 7, the I/O devices such as the keyboard and the display and the processor may located at different positions and connected by cables, wireless transmission or similar or be part of a local area network or interconnected by telecommunications networks.

Fig. 2 is a schematic representation of an ICC 5 in accordance with the present invention. The present invention is however not limited thereto. The ICC 5 includes at least an input/output (I/O) port 10 and some permanent storage, e.g. a non-volatile memory which may be provided, for instance, by a an EEPROM 15 connected to the I/O port 10 via a bus 17 or by battery-backed random access memory (RAM). The I/O port 10 can be used for communication with the terminal 1 via card reader 7. An integrated circuit card is a card into which one or more integrated circuits are inserted to perform at least memory functions. Optionally, the ICC 5 may be a self-contained portable intelligent card and include a read/writable working memory e.g. volatile memory provided by a RAM 14 and a central processor 12 as well as all necessary circuits so that card ICC 5 can operate as a microprocessor, e.g. read only memory 13 for storing code, a sequencer 16 and connections with the card reader 7 for receiving the voltage sources Vss and VDD, a reset for the processor 12 and clock CLK to the sequencer 16. In accordance with the present invention the ICC 5 can be used as bank card, credit card, debit card, electronic purse, health card, SIM card or similar.

The present invention provides an integrated circuit controlled transaction management system intended to execute between an ICC 5 and a terminal 1 connected or not connected to a central unit, a transaction consisting of at least one execution of the following sequence:

1. creating a communication link between the ICC 5 and the terminal 1;
2. performing a compatibility check to ensure that the ICC 5 and the terminal 1 are mechanically and electrically compatible;
3. selection of an application including selection of a computer program and the associated data set that defines the transaction in terms of the specific ICC 5 and the terminal 1 combination involved;
4. execution of the application;

5. termination of the transaction, which optionally includes breaking the communication link between the ICC 5 and the terminal 1; whereby an interpreter is used to execute the application, either on the ICC 5 or on the terminal or on both. A transaction is an exchange of at least data between two or more devices and in accordance with the present invention is not specific to a commercial financial transaction. Such a system is known from PCT/BE 95/00017. The ICC 5 may be a mere memory ICC, i.e. not including processor 12, and the ICC 5 undergoes the transaction as determined by the terminal 1. Alternatively, the ICC 5 may be a self-contained portable intelligent card and the transaction may be determined by the terminal 1, by the ICC 5 or by both. In accordance with the present invention the ICC 5 may include program code to securely enhance a terminal's processing. In particular, ICC 5 may beone or more maintenance cards which may be used to update the applications stored in the terminal 5.

In accordance with the present invention software is run in the terminal 1 and optionally in the ICC 5 in terms of a "virtual machine". The virtual machine (VM) in accordance with the present invention explicitly makes available a theoretical or virtual microprocessor with standard characteristics that define addressing mode, stack usage, register usage, address space, addressing of I/O devices etc. in a generic way. The kernel for each particular CPU type used in a terminal 1 or ICC 5 is written to make that respective processor 2, 12 emulate the VM. It is a specific aspect of the present invention that the kernel for the VM provides drivers for I/O devices and all low-level CPU logical and arithmetic functions, flow control, time handling. The provision of I/O drivers in the VM has the advantage that any program written for the VM in accordance with the present invention addresses standard virtual I/O devices. The implementation of the VM on a particular CPU then provides for the physical I/O devices connected to the terminal 1 or ICC 5 to be behave like the addressed virtual I/O devices. The VM in accordance with the present invention is very compact and has been implemented successfully on a Siemens SLC044CR chip (derivative of the INTEL 8051 family) which may be included on an ICC 5. The VM in accordance with the present invention makes a high degree of standardization possible across widely varying CPU and I/O types, and simplifies program portability, testing, and certification. In accordance with the present application such a VM will be described as a portable virtual machine 20. The portable VM 20 includes a virtual microprocessor and a driver for an I/O device. The VM 20 provides logical and arithmetic functions and addressing of memory and the at least one input/output device. The portable VM 20 in accordance with the present invention provides program portability across

heterogeneous terminals 1 and card 5 by treating terminal and/or card programs as the intermediate code of a compiler. This code consists of streams of byte-codes, called tokens. Terminals 1 or ICC's 5 then process this code by interpreting it or by other means such as native code compilation. Virtual machine interpretation of the tokens may preferably be accomplished by one of three methods: directly interpreting virtual machine instructions, translating the virtual machine language into a directly executable intermediate form, or just-in-time compiling it into actual code for the target CPU. The latter two methods offer improved performance at a modest cost in complexity. Tokens are provided in a set which may be looked upon as the set of machine instructions for the VM 20.

Application programs in accordance with the present invention are embodied as modules which may include a list of tokens as executable code. In accordance with the present invention there are two basic types of modules: executable modules, which have an entry point that is called directly by the VM 20 when the module is loaded, and library modules, which act as resources to other modules by providing exportable procedures that may be executed individually by intermodule calls.

The token set in accordance with the present invention includes firstly, the instruction set of the VM 20, which provides the instructions expected for a general processing language and are required for the efficient execution of programs and secondly tokens which provide what are normally called "operating system functions". In terminals 1 or card 5, operating system functions include in accordance with the present invention specific functions such as I/O drivers, e.g. for displays and keyboards, and in terminals 1 and card 5, system functions may also include management of data object communication and transmission through I/O ports, and also inter-module access and access control mechanisms. Tokens are preferably provided for the operation system of the VM, for stack manipulations, for socket handling, for control, e.g. exception handling, for the sockets themselves including their access rights, for I/O device access, for time handling, for language and message handling, for handling I/O readers, e.g. ICC, magnetic stripe card and modem handling, for blacklist management, for security algorithms, for terminal services, for database services, for data object handling, e.g. TLV handling, for module handling and for extensible memory handling,

Single-byte tokens are referred to as *primary* tokens; these refer to primitive instructions such as are commonly found in any instruction set. Multi-byte tokens are referred to as *secondary* tokens, and are used for less-frequently-used services. The

complete token set for the VM 20 is provided in the appendix. As shown schematically in Fig. 3 an application program is written on a PC host development system 70 and debugged and type approved in an appropriate high level language such as Forth, C, Pascal etc. Then the source code of the program is compiled on a token compiler 71 into a stream of tokens. This token stream is separately combined with other data (the corresponding in-line data) needed by the program and a header, and encapsulated in a module delivery file to create a module 72, preferably in a standardized Module Delivery Format. If the module contains executable tokens it is delivered in the Executable Program Format. It is a particular aspect of the present invention that the executable modules contain not only the stream of tokens but also all corresponding in-line data (encapsulation). It is a particular further and separate aspect of the present invention that the modules in accordance with the present invention contain an indication of how much read/writable memory should be allocated by the VM 20 for the module to execute.

Module 72 is delivered to the terminal 1 by any suitable means, e.g. in an ICC 5, via a telecommunications network. After a module is downloaded, it is stored in a "module repository." When its function is required, the terminal 1 or card 5 uses a *token loader/interpreter* 73 to process the tokens for execution on the terminal's CPU 2. This process consists of executing the function associated with each token. The token loader/interpreter 72 is provided by the VM 20 in accordance with the present invention.

The VM related software on a terminal 1 can be divided into four major categories:

- *The Kernel*, which includes terminal-dependent implementations of I/O drivers and all functions required in this specification for supporting the VM 20. Every other software related to the VM 20 is written in machine-independent tokens..
- *Terminal Resident Services* (TRS) are at least one module which runs on the VM 20 as an applications manager, and includes all non-application functions, libraries supporting these functions, module loading functions, and the main loop defining the terminal's behavior. Special tokens (e.g. **(DIOCTL)** allow terminal-dependent aspects of I/O devices to be defined as in-line functions.
- *Terminal Selected Services* (TSS) include applications such as payment service functions, and libraries supporting these functions. TSS contain only terminal independent tokens and are resident on the terminal 1. The TRS main program loop will select and call TSS functions as needed by a particular transaction.
- *Card Selected Services* (CSS) include functions supporting terminal transactions, such as payment service functions that are used as part of a TSS application. CSS are resident on

an ICC 5 and downloaded to a terminal 1 as requested. For terminals 1 with two ICC readers 8 (e.g., one for normal transactions and one for maintenance) there may be two independent sets of CSS (CSS1 and CSS2).

All software on a terminal 1 in accordance with the present invention, above the kernel, is organized as a set of separate *modules*. The fundamental characteristic of a module is that it is a collection of definitions or program functions that have been passed through a token compiler 71 and encapsulated as a single package for delivery to a target environemnt, e.g. a terminal 1 or an ICC 5. The main Terminal Program (TRS), each application, each library, and each CSS download are examples of modules. Preferably, all modules use a standard format. The kernel in a system according to the present invention defines the VM 20 which provides a variety of high-level services to these modules, such as:

- general purpose CPU and instruction set, represented by tokens;
- general purpose I/O support for common devices, with provisions for generic I/O to support additional devices that may be added;
- database management functions;
- data object transmission management, including format conversions and other functions;
- management of token modules, including maintaining them in storage (updating as necessary) and executing them on demand. In a preferred embodiment of the present invention, execution of modules remains in the control of the VM 20 at all times. Hence, modules never take control of the host processor but are merely passive with respect to the VM 20. Preferably, the VM 20 always operates in supervisor mode and can only execute instructions defined in terms of its token set and no module may operate in user mode, i.e. assume control of the VM 20. Hence, in an implementation of the VM 20 using a memory map, only one memory map is created , namely the supervisor map.

No module may compromise the operating system defined by the VM 20. This is guaranteed because a module may only contain tokens from the VM token set and none of these tokens allow access to the code space where the kernel is stored. If a token is encountered by the VM 20 which lies outside the defined set, an exception is thrown (**ILLOP**).

As shown schematically in Fig. 4, terminal 1 includes the terminal specific operating system 8 responsible for loading the TRS module of the VM 20 (loading procedure is described later). Code for the VM 20 is stored in the terminal read only non-volatile memory 11. Before loading the TRS, the terminal volatile memory 19 of the

terminal 1 is empty of all transaction related data, the terminal read/write non-volatile memory 18 contains the applications to be executed by the VM 20 in the form of modules 72 in a module repository, the non-volatile databases containing user specific data and libraries such as the plug library which will be described later. If the VM 20 is also implemented upon an ICC 5, the same principles as described above apply with respect to the read only non-volatile memory13, the volatile memory 14 and the read/write non-volatile memory 15 of ICC 5.

As VM 20 is a virtual machine it addresses all forms of terminal or ICC memory 11, 18, 19; 13, 14, 15 as virtual memory, i.e. they are all addressed in logical address space as seen from the VM 20. In an actual implementation of VM 20, these logical address spaces are mapped into actual address spaces in the memory of terminal 1 or ICC 5 by a memory map or similar. In the following, reference will be made to volatile, read/write non-volatile and read only non-volatile memory as being part of the VM 20. It is to be understood that this refers to logically addressed memory space unless specific mention is made to actual addresses in an implementation of VM 20 on a terminal 1 or ICC 5. Volatile memory does not survive program loading or power-down and/or rebooting. Preferably, volatile memory does not survive power down for security reasons. Non-volatile memory survives program loading or power-down and rebooting.

Virtual Machine description

A schematic representation of the VM 20 in accordance with the present invention is shown in Fig. 5. Preferably, the VM 20 is a stack machine and has a data stack pointer (stored in the data stack pointer register 32) that points to a pushdown data stack 27 preferably in on-chip memory. All operations are performed on this stack 27. The data stack 27 is used to contain parameters of procedures and temporary results from expression evaluation. Data is pushed onto or popped from the data stack 27. The VM 20 also includes a return stack 28. The return stack 28 may be used by the VM 20 to contain return addresses, and also may be used for temporary storage. This multiple stack architecture is known from the Forth programming language (ANSI X3.215-1994). This architecture has been further modified for portability, code density, security, ease of compilation, and for use with other programming languages. For example, it contains provisions for local variable frames used in high level programming languages, such as C. Thus, token compilers 71 in accordance with the present invention can be written not only for Forth but also for C and other programming languages.

In accordance with the present invention there is only one data and return stack 27, 28 associated with the VM 20. Data and return stacks 27, 28 are not created for each processing thread. In accordance with the present invention application programs may only retrieve from the return stack 28 what they have explicitly put on it inside the current procedure, and must remove data placed on the return stack 28 during the current procedure before exiting the procedure. Optionally, the VM 20 includes a security manager for providing run-time integrity and which monitors every activity on the return stack and checks all data is removed during the currant procedure. In addition to data placed there explicitly for temporary storage, the VM 20 may hold exception execution state information, frames for local variables, loop control parameters and database context on the return stack 28.

In accordance with the present invention, the VM 20 may include a plurality of stacks. For example, rather than use only the return stack 28, optionally, further stacks 29 may be provided such as exception and frame stacks. An exception stack is used to store the execution state during an exception processing. A frame stack is used to maintain local variables information, as well as C language-like activation records. As mentioned above, the exception and frame stacks 29 may be implemented on the return stack 28.

The data, return stacks 27, 28 and other stacks 29 such as the exception stack are not in a memory space directly accessible by any application program. The data or return stacks 27, 28 cannot be addressed directly, they are only accessible via stack operations. For security reasons, there is no limitation on how the data is stored in an actual implementation, so that malicious persons may make no assumptions as to how the data is stored physically.

The VM 20 includes a virtual central processor unit 22 which includes a virtual arithmetic logic unit 23. The VM 20 addresses a virtual or logical data space 24 and which is treated by the VM 20 as random access memory (RAM) and would normally be implemented as part of the volatile memory 19, e.g. in RAM in an actual terminal 1 or card 5. The logical data space 24 is accessible for data storage only. The token stream of a module is stored in by the VM 20 in code memory 26 which is treated as read only memory (ROM) by the VM 20. Only the VM 20 may read the tokens. Code memory 26 is not accessible for application programs nor by any token.

The VM 20 may include virtual registers 30. The VM 20 registers 30 may include Token Pointer register 31 which holds the pointer which points to the next token to be executed, Data Stack Pointer register 32 which holds the pointer which points to the

current top of data stack 27 (TOS) location, Return Stack Pointer register 33 which holds the pointer which points to the current top of the return stack location, Frame Pointer register 34 which holds the pointer which points to the frame start in data space 24, Frame End Pointer register 35 which holds the pointer which points to the frame end in data space 24. No direct access is provided to registers 31 to 36 by the token set but only via the register access tokens.

Finally, the virtual I/O devices 25 are linked to the CPU 22 by a virtual bus system 21 which also links stacks 27-29, ALU 23, registers 30, code memory 26, and data space 24.

VM 20 in accordance with the present invention is defined preferably as a byte-addressed, two's complement, 32-bit machine, with 32-bit registers and stack elements however the invention is not limited thereto. The register/stack size is referred to as the *cell size* of the VM 20, a cell being the basic unit of manipulation on the stacks and by the VM registers 31 to 36. The size of the cell used by the VM 20 is not considered to have a material affect upon the present invention.

All programs run from modules must observe the following rules.

- Programs may not assume that the data in any of the above categories is guaranteed to be held on the return stack 28, and so may use only the retrieval mechanism specified explicitly for a given category as the security manager ma delete any such data.
- Programs that pass data in one of these categories to the VM 20 must take appropriate action to recover the data storage from the VM 20 before exiting from the procedure in which it is passed otherwise the VM 20 will deallocate the relevant part of memory.
- Programs must assume that any data previously placed on the return stack 28 is rendered inaccessible by passing data in one of these categories to the VM 20.
- Programs must assume that any data in one of these categories passed to the VM 20 is rendered inaccessible by executing code that places values on the return stack 28, until such time as those values are removed because the return stack is only accessible by the specified token procedures.

In accordance with the present invention the operating system of VM 20 defines a single address space 24 available to each module. This address space 24 shall be accessible for data storage only, and will be referred to as "data space" 24.

Data space 24 is divided into three and one optional logical regions, each of which is individually contiguous:

1. *initialized data space* 41, which contains initial values specified at compile time and set

when the VM kernel is activated and subsequently when a module containing initialized data is loaded;

2. *uninitialized data space* 42, which contains variables and static buffers allocated during program compilation. This data space 42 is initialized to zero by the VM 20;

3. *frame memory* 46, which is managed by the frame tokens;

4. optionally: *extensible memory* 45, which contains one or more buffers allocated dynamically during program execution.

There are two additional data regions, which are not directly addressable:

5. *extended memory* 43 typically mass storage which is used to contain data objects and volatile databases;

6. *non-volatile memory* 44 is used to contain data that is guaranteed by the VM 20 to survive module loading or power-down and rebooting (within the limitations of the terminal hard-ware), including the module repository and non-volatile databases. This may be implemented in battery-backed RAM, disk, or other persistent storage. Non-volatile memory 44 may be implemented as part of the read/write permanent storage 18.

To increase the security of data, extended and non-volatile memory 43, 44 is accessed only through tokens that provide "windows" to selected data in the form of buffers in uninitialized data space 42. Hence, a programmer can only ask for a record and cannot know the exact location of data in order to access it. Preferably, there is no file nor tree structure which allows a programmer to locate personal files or databases.

Within each data space 24 used by a module, memory is allocated by relative addressing and only at run time. This means that an address only has meaning within each module when it is loaded. As absolute addressing is not used, it is not possible for a module to gain access to data of another module except by the secure module access mechanisms described later.

The *frame* mechanism allows the VM 20 to satisfy the requirements of languages such as C that permit local variables to be defined at run time. A frame holds procedure parameters passed on the data stack 27 as well as "local" variables (temporary data storage that shall be freed automatically when the frame is released, normally at the end of procedure execution). Frame start and end pointers are maintained automatically by the VM 20 within the frame. The frame pointer register 34 points to the logical base of the frame and frame end pointer 35 points to the logical end of the frame in data space 24. Parameters can be fetched from the frame using the frame access tokens.

The VM 20 may optionally provide a dynamically allocated pool of extensible

memory 45 as a single extensible buffer managed by the VM 20 which appears outside the program's uninitialized data space 42. Programs may request an allocation of a specified amount of extensible memory 45 and are returned a pointer to a base address for that memory 45. Subsequently programs may release memory 45 from a given address, e.g. on termination of the program, causing *all* allocations beyond that address to be released.

It is preferred if modules are executed in a single-thread but the invention is not limited thereto. This means that if one module calls a second module, the second module terminates and all resources of the second module are deallocated before the VM 20 returns to the first module and continues processing. Fig. 6 show a schematic representation of logical memory as seen by the VM 20. As shown schematically in Fig. 6, a first module (on the left) with initialized memory 41, uninitialized memory 42 and frame memory 46 and token code space 26 has been loaded into the read/writable memory starting at address 1. The first module has also called and been allocated a portion 45 of the extensible buffer. When the second module (on the right) is called by the first module (e.g. to import the function fgh which is in the exclusive list in the header of module 1 of functions which may be imported), data space 24' including initialized memory 41', uninitialised memory 42' and frame memory 46' are allocated as required starting at address 2. The tokens of module 2 are read directly by the VM 20 from the module repository which is an option allowed in accordance with the present invention. If called from module 2, the extensible memory 45' for the second module is allocated by the VM 20 higher in the memory than the extensible memory 45 for the first module. When the second module is completed, all memory above address 4 is deallocated ("rubber band effect"). Preferably all temporarily stored data is erased on deallocation. If necessary, more extensible memory 45 could then be called on return to the first module. If the second module is then called again, it will be allocated a different starting address for the extensible memory 45' than for the first time it was called.

All tokens except the extensible memory management tokens **EXTEND**, **CEXTEND**, and **RELEASE** and the exception handling tokens **THROW** and **QTHROW** are required to have no net effect on the extensible memory pointer. If a token allocates extensible memory 45 it must also release it, including any effect of cell-alignment. Successive allocations of extensible memory 45 are preferably contiguous within a module but need not be contiguous between modules, except that inter-module calls using **IMCALL** or **DOSOCKET** tokens shall preserve contiguity. An automatic release of dynamically allocated extensible memory 45 shall occur when a module's

execution is completed, limiting the effects of program failure to release memory cleanly. In addition, if a **THROW** exception occurs, the allocation of dynamically allocated extensible memory 45 may be restored to its condition at the time of the governing **CATCH** exception.

User variables are cell-sized variables in which the VM 20 holds contextual information for the client programs. Storage for user variables is pre-allocated by the VM 20. A limited number of variables may be provided, e.g. sixteen variables (referenced as 0 to 15). An implementation of VM 20 that supports multitasking may provide one set of user variables for each task.

The VM 20 provides a single exception handling mechanism via the tokens **CATCH**, **THROW** and **QTHROW.** These tokens are derived from the Lisp exception handling mechanism, and appear in ANS Forth as **CATCH** and **THROW**. The purpose of this mechanism is to provide for local handling of exceptions under program control at various levels in the software. The concept is that the program passes a function's execution pointer to the token **CATCH**, which will execute that function and return a code indicating what, if any, exception occurred during its execution. **CATCH** records implementation-dependent information sufficient to restore its current execution state should a **THROW** occur in the function passed to **CATCH** for execution. This includes (but is not limited to) data and return stack depths, the frame pointer and, in some cases, the extensible memory pointer. The collection of information representing an execution state is referred to as an "exception frame." Exception frames are kept on the exception stack. Following a **CATCH**, the program can examine any exception code that may have been returned, and elect to handle it locally or **THROW** it to a higher level for processing. The VM 20 provides a default outermost level at which exceptions will be trapped. This outermost level will be activated when no inner level of **CATCH** has been established. The default exception handler aborts any current terminal transaction and attempt to reload TRS modules and re-enter the TRS main loop. The VM 20 throws ANS Forth exception -10 (Division by zero) should that condition occur. The VM 20 may throw other general exceptions supported by ANS Forth, e.g. as given in the attached appendix.

For handling devices and input/output (I/O) services, each device, including those devices whose lower-level operation is hidden by the VM 20 behind device-specific functions, is assigned a device type (used to categorize result codes) and a unique device

number. Device numbers are arbitrary; however, references to device numbers -1 through 15 (4 bits) may be made with only a single token, and so these are assigned to the most common devices such as keyboards, ICC readers, magnetic stripe card readers, displays, printers, power management devices or vending machine. General I/O facilities are provided by functions taking the device number as an input parameter.

A terminal 1 in accordance with the present invention preferably contains at least three major non-volatile databases: an application-specific transaction log, a database of messages in one or more languages and a database of modules. The VM 20 protects databases as much as possible as they may contain private information. Access to databases is restricted. The VM 20 provides a mechanism for handling databases (the VM 20 as a "server") that conceals implementation details from the application software (application as the "client"). No direct access to the database is allowed from a module running on the VM 20. The services implement the following features which will be described with reference to Fig. 7:

- At any given time the client, i.e. a program running in an module, has access to one currently selected database (**DBCURRENT**) and one currently-selected record number (**DBRECNUM**) that is shared across all defined databases.
- Information about each database is transferred between client and server through a Database Parameter Block (DPB) 51, which the server can access for reading and writing. The client "owns" the DPB 51, in the sense that it is in the client's data space 24; but the client is not allowed to access it directly. Instead, only database service tokens may be used to access the data. The DPB 51 has a standard structure including fields for at least a DPB link, a database pointer, an indication of the database type and of the record size and the next available record number. All information to specify a database must be preset in the DPB 51. Client software may not make any subsequent direct access to the DPB 51 and must make no assumptions about the values directly held in the DPB 51 after the module defining that DPB 51 has been loaded for execution. Database Parameter Blocks 51 are passed to the token loader/interpreter in the form of a pointer (DPB link) to a linked list in the initialized data section of a module. This field must be preset to the address in initialized data of the next DPB 51 in the list; or zero if this DPB 51 is the last or only DPB 51 in the list. For compiled databases that exist in the client's initialized data space, the DB pointer must be preset to the "origin" address in initialized data. For databases whose storage is controlled by the server, the field must be preset to zero. The DB type provides details of the database in coded form. There are at least three kinds of databases:

"Volatile" databases whose content does not need to be preserved between module loads or across a power-down of the terminal 1 on which it resides.

"Non-volatile" databases whose content must be preserved between module loads or across a power-down. If the module defining a non-volatile database is replaced, the database is destroyed when the old module is unloaded.

"Compiled databases" are constructed by the token compiler 71 in a contiguous area of initialized data as fixed-length records. Records may not be added to or deleted from a compiled database, and the database may not be initialized at run-time, but otherwise full read-write capability shall be supported.

The next available record number field must be set to one plus the last allocated record number in the database for compiled databases. For any other database this field is set to zero.

• The address of a window onto the current record (a record buffer 53) is provided by the server to the client for each client database. For certain database operations the client may pass the addresses of strings and key buffers 52 to the server. For each database that has been made known to the server by a client module, a record buffer 53 is provided by the VM 20. This buffer 53 starts at an aligned address. The content of the record buffer 53 associated with a particular database after a record selection remains available until the client selects another record from that database. Except for these record buffers 53, compiled databases, and parameters passed on the data stack 27 by specific database functions, no other data space 54 is shared between client and server. Programs may not assume that records in a database are contiguous in memory.

• A database is instantiated by the loading process for the module in which its DPB is defined. Volatile databases installed by application modules are uninstantiated automatically and transparently by the server when the application is terminated by Terminal Resident Services, when all server-allocated data storage relating to those databases is released.

• The server deletes non-volatile databases when the module that defined them is replaced. If the module is loaded when replaced, e.g., in the case of a TRS module, the server must delete the module's non-volatile databases when the module is unloaded.

The action taken by the VM 20 when a database is instantiated at module load time depends on the value of **DB type** and **DB pointer** in the DPB 51, and whether the database is volatile or non-volatile. If the database is a non-volatile type, the DPB address is used in conjunction with the module identification (module ID) to identify any prior data

that belongs to the database. If prior data exists, the next available record number is restored to its previous value. Otherwise, the server (VM 20) instantiates new non-volatile storage space and sets the next available record number to zero. In both cases a buffer 53 is provided for the current record in the database. If **DB pointer** is zero and **DB type** is not a compiled type, then the server instantiates or makes available the storage required for the database, initializes the storage to all zeros, provides a buffer 53 for the "current record" of the database, and sets the next available record number (**DBAVAIL**) to zero. If **DB pointer** is non-zero and **DB type** is a compiled type, then the server sets up internal structures to make use of the client data structure whose origin address has been passed at **DB pointer** and sets the next available record number (**DBAVAIL**) to the value passed in the next available record number field of the DPB 51. The server maintains the actual database records 55, the relationship between the address location and the record in a database control block 56 and a record of which database a module is currently accessing in a context info block 57.

The secure module handling procedure in accordance with the present invention will now be described with reference to Fig. 6. In Fig. 6 an area of logical read/write memory is shown. An area of memory which may be accessed by the left hand module (first module) has a dotted line. An area of memory which cannot be accessed by the first module has a border of a continuous line. An area of memory which can be accessed by more than one module is shown with a dot - dashed line. VM 20 protects databases DB1 and DB2 and the database repository and the modules in the module repository so that they are not accessible by any module. After the first module is loaded, the uninitialised data in memory 42 can be accessed by the first module but the VM 20 does not allow that any area outside this module may be accessed directly by the module. Access to registers, stacks or frame memory 46 can only be made through the relevant tokens. Databases can only be accessed via the window procedure mentioned above. In particular the first module cannot access its own program memory 26 where the tokens are located nor access any other module. This is of importance for protection against viruses. In accordance with the present invention the memory is allocated to first module is defined and protected. It is defined by the allocation of memory in accordance with the indication of the amount of memory to be allocated contained within the module. It is protected because no other module may access the allocated space, and no other loading mechanism is provided for any program other than for modules. As the preferred method of running modules is monothreaded, any memory allocated in the extensible buffer 45 is deallocated before any

other module may become active. Deallocated memory is preferably erased.

The exclusive import list of the first module is in its header which the first module cannot access directly. The VM 20 reads the header and calls the second module which is mentioned in the import list (function fgh from the second module). The VM 20 loads the second module and allocates memory for the uninitialized data 42', frame memory 46', and initialized data 41'. The first module cannot access any part of the second module and vice versa. In the header of the second module, the function fgh has been placed in the exclusive list of functions which can be exported. This makes the function fgh available for other modules. The VM 20 searches for the function fgh in the code memory space 26' of the second module and executes the token stream with corresponding in-line data (represented by the stream TITTITT). In this example, this piece of code requires access to a database DB2. A database in accordance with the present invention is "owned" by a module, i.e. a database can only be accessed by the module which instantiated it on first loading of the module. The database access tokens read from code space 26', are executed by the VM 20 which allocated a buffer 53' in the uninitialised data space 42' of the second module on loading. The function fgh requires access to the third record of DB2. The VM 20 then transfers the referenced record to the window 53' in the second module from which it is exported to the uninitialised space 42 of the first module. Using the same database window procedure, the first module may also obtain a record from its own database DB1 which is transferred to the buffer 53 in the uninitialised data space 42. The first module may now operate on the results of the two procedures.

The VM 20 preferably deals with data objects by means of Basic Encoding Rules or Tag, Length, Value (BER-TLV shortened to TLV for this application), as described in ISO/IEC 8825 (1990). TLV data objects consist of two or three consecutive fields: a *tag* field specifying its class, type and number, a *length* field specifying the size of the data, and if the length is not zero, a *value* field containing the data. Because ICC card responses are generally limited in size to say 255 bytes or less, there is a maximum size of a TLV object in accordance with the present invention. The tag field is preferably one or two bytes, the length field is preferably one or two bytes, and thus the maximum size of the value field is preferably 252 bytes (a field this long requires two length bytes, as explained below). The first byte of the *tag* field is broken into three fields. Bits 7 and 8 specify the class of the object. Bit 6 determines if the value field contains "primitive" data or if it is a "constructed" object consisting of other TLV-encoded fields. Constructed objects are also called *templates*. They cause their value fields to be parsed for TLV sequences when they

are encountered. Bits 1 to 5 specify the number of the object or, if all these bits are set, they indicate that additional tag bytes follow. The additional tag bytes have their eighth bit set if yet another byte follows. All bits in up to two bytes are used to determine a tag name. The *length* field consists of one to three consecutive bytes, typically two. If bit 8 of the first byte is 0, then bits 1 to 7 indicate the size of the value field. If bit 8 of the first byte is 1, then bits 1 to 7 indicate the number of bytes that follow. The following bytes, if any, indicate the size of the value field and occur with the most significant byte first. The *value* field can consist of "primitive" data or be "constructed" with additional TLV encoded sequences. If bit 6 of the first byte in the tag field is set, then the value field contains additional TLV sequences. The primitive objects can be encoded in several different formats: Binary Coded Decimal nibbles with leading zeros or trailing nibbles with all bits set, binary numbers or sequences of bytes, character sequences of alpha/numeric or ASCII bytes, or undefined formats. Each is handled differently as it is used. An ICC 5 may also use a Data Object List (DOL) to request values of specified tag names. The card 5 sends a DOL consisting of a list of tag and length fields, and the terminal 1 returns the corresponding value fields, without delimiters.

Each TLV to be used must be defined by the terminal or application programs to establish its data type and name. Since the terminal program and the application programs are developed separately, the VM 20 in accordance with the present invention uses a linked structure (a balanced binary tree) to allow rapid addition and removal of tag names from the global tag name list. This requires that the following structure be compiled for each TLV in initialized data space 41 in the module defining the TLV:

Link A cell with "left" (high-order two bytes) and "right" (low-order two bytes) components providing links to elements of the tree.

Link left A 16-bit signed offset from this TLV's access parameter to the access parameter of a TLV record whose tag is numerically less than this record's tag. A value of zero indicates that this TLV is not linked to a TLV with a tag numerically less than this one.

Link right A 16-bit signed offset from this TLV's access parameter to the access parameter of a TLV record whose tag is numerically greater than this record's tag. A value of zero indicates that this TLV is not linked to a TLV with a tag numerically greater than this one.

Tag A two-byte string whose big-endian numeric value is the TLV tag.

Type A single byte that specifies control information.

Reserved A byte that must be initialized to zero by the compiler 71.

Data A cell that holds VM-specific information including access to the length and value fields of this TLV. This field must be initialized to zero by the compiler 71. The system must also maintain a status byte for each TLV. This may be the Reserved byte in the above structure. The low-order bit of this byte shall be set if the TLV has been assigned a value as a result of being in a sequence that has been processed by the tokens **TLVPARSE** or **TLVSTORE**. The purpose of maintaining an assigned status is to identify TLV values that contain valid data (which may be zero) and distinguish them from TLV values that have never been set and are therefore invalid. The VM kernel manages a global list of TLV tags by maintaining a list of pointers to the initialized data space 41 containing their actual definitions as described above. When a module is loaded, its TLV definitions are added to this list as part of its initialization; when it is unloaded, its TLV definitions shall be removed from the list automatically by the VM 20. An exception may be thrown if the module contains a TLV definition that already exists. The address of the **Link** field described above is returned as the "access parameter" for TLV references. The programmer should not access these fields directly, nor make any assumption about their contents, after the VM 20 has instantiated the TLV definitions.

References to TLV definitions in the source code are compiled as either direct references to the definition structures defined above, or numerical tag values. Within certain binary TLV definitions, individual bits or groups of bits are defined to have certain meanings. These are referred to as "TLV bits". References to TLV bits may be compiled with a literal pointing to a bit within the value field of the TLV. Bit 0 is the least significant bit of the first byte, bit 7 is the most significant bit of that same byte, bit 8 is the least significant bit of the second byte and so forth.

The data assigned to a TLV definition is exposed to the application through a 252-byte scratch pad area maintained by the VM 20 in the form of a database window (see Fig. 7). The application program is permitted to change the contents of this scratch pad area. If changes are to be retained, an address and length within the scratch pad area must be passed back to the **TLVSTORE** token. The address and contents of the scratch pad area may become invalid when any TLV token is subsequently executed.

As part of the security management of cards 5 introduced into reader 7, a check for cards 5 which are known to be lost or invalid is made. A list of such cards 5 is known as a black or hot card list. Black or Hot Card List management is provided by a set of

dedicated functions that are specific to the management of a large hot card list. A typical list may contain 10,000 primary account numbers (PAN) of up to10 bytes each, or 20 binary coded decimal (BCD) digits. The PAN entries are stored in compressed numeric (cn) format, right padded with hexadecimal FH 's. As a PAN is a maximum of nineteen BCD digits, an entry in the list will always be padded with at least one FH . When searching in the hot card list, FH 's in a list entry are considered as wild cards or "don't care" digits, but any FH 's in the PAN used as input are not wild cards. Wild cards can only appear at the right-hand end of an entry. A PAN shall be considered found in the hot card list if there is a list entry that is identical up to the first FH in the entry.

Another part of security management is the provision of cryptographic services for encrypting and decoding data. Any suitable encryption methods may be used. Three cryptographic algorithms are particularly provided for the VM 20: modulo multiplication and modulo exponentiation, which are used in the RSA algorithm; and the secure hashing algorithm SHA-1, but the invention is not limited thereto. Modulo Multiplication performs a multiplication of two unsigned values x and y, where the product is reduced using the modulus z. The formula is:

$$result = mod(x*y,z)$$

The input values (x,y,z) are all the same length. They are represented by byte strings and can be any multiple of 8 bits up to and including 1024 bits. The values must be in big-endian byte order.

The Secure Hash Algorithm (SHA-1) algorithm is standardized as FIPS 180-1. SHA-1 takes as input messages of arbitrary length and produces a 20-byte hash value. Modulo Exponentiation raises an unsigned value x to a power given by an unsigned exponent y, where the product is reduced using the modulus Z. The formula is:

$$result = mod(x^y,z)$$

The input value x and modulus z are represented by byte strings and may be any multiple of 8 bits up to and including 1024 bits. The values must be in big-endian byte order.

Services and therefore software and even I/O devices may change with time depending upon market requirements. When major changes are required, an update of the software in the terminal 1 may be carried out by hand or remotely via a telecommunications network. However, for user dependent services it is preferable to have a dynamic but secure method of making minor or user-specific upgrades to the services provided by a terminal 1. The "plug and socket" software mechanism in accordance with the present invention provides a flexible and secure way of on-line configuration of the

different modules that make up terminal programs and applications. As shown schematically in Fig. 8, in the transaction system in accordance with the present invention, a number of procedures (referred to as 'sockets' 60) may be defined that may be inserted by the application programmer (and hence secure because it is under acquirer control and under payment system supervision) into an application 61, 62 to act as placeholders for the addition of additional enhancing code (referred to as "plugs" 66) during transaction processing. All additional code to be plugged into a socket 60 must be written in terms of the token set of the VM 20. Sockets 60 are preferably placed at various suitable points in existing terminal applications 61, 62 or even in the terminal program itself. They are used to refer to library functions and may even occur inside a library function if a payment system foresees the need to change the way a library function operates. Sockets 60 are initialized to default behaviors by the VM 20. If no further action is taken by the terminal program, the default behavior of sockets 60 will be to do nothing when they are executed (i.e. no-operation).

Plugs 66 include executable code, written in tokens supported by the terminal 1, that may be inserted at points defined by sockets 60 to enhance the default terminal logic. Plugs 66 may already exist in the terminal 1 in a plug library 63 to be invoked from the ICC 5, e.g. socket/plug identifiers 67 in an ICC 5 and logic in the terminal 1. Socket/plug identifiers 67 include a refernce to both the plug and the socket to be plugged whereby the plug is not on the ICC 5 but in the library 63. Plugs 66 may also come from an input device 65 (such as the ICC 5 or a host system connected to the terminal 1), but only if agreed by the members of the payment system, e.g. issuer, acquirer, and merchant.

At the conclusion of a transaction, the sockets 60 are restored to their original application default behaviors. In accordance with the present invention it is preferred that an ICC 5 does not contain entire applications but only plugs 66 that enhance existing terminal applications as these require less memory.

Sockets 60 hold execution pointers, also known as procedure pointers, that allow the creation of a procedure whose behavior may be changed at execution time. Sockets 60 may be viewed (and implemented) as an array of procedures that are accessed through the **DOSOCKET** token, which takes the socket number as an in-line byte, or by the **IDOSOCKET** token, which takes the socket number from the data stack 27.

Sockets 60 enable re-configuration of a terminal program or application to provide variations or enhancements in the transaction processing flow. Alternatively, sockets in ICC' 5 may allow upgrading of ICC's 5 from a terminal 1. Sockets 60 provide an interface

between software modules and procedures that may be coming from several different sources (acquirer, issuer, etc.). Since an acquirer and an issuer have a contractual relationship, they may agree to use specific sockets 60 provided by the acquirer's program in a terminal so that an issuer may extend the behavior of the program, for example to provide a loyalty function (air miles, coupons, etc.).

A module may specify that sockets 60 be reconfigured automatically when it is loaded for execution, or a client program may programmatically assign a new procedure to a socket at run-time. Provided security conditions permit it, sockets 60 in an application may be assigned a default behavior and then may be re-plugged with new procedures by subsequent modules, in order to provide specialized behaviors. To avoid indefinite situations it is preferable if all procedures vectored to use a particular socket 60 have no data stack effect (except for socket zero as explained later). This ensures program continuity no matter which vectored version of the procedure is executed. The default action of all sockets 60 before modification shall be at least that of a no-operation.

An acquirer may allow transaction enhancements by code on an ICC 5 as part of the CSS referred to above. If so, they may be implemented with sockets 60. A library or executable module may include the definition of new sockets 60 for later plugs 66 coming from an ICC 5. In this case the module would define a socket 60 and then use the **SETSOCKET** token to assign a default behavior to it (often a null behavior). If access control allows it, an ICC 5 could later down-load a plug 66 including tokens that define a new behavior and then use the **SETSOCKET** token to store it in this same socket 60, overriding the default behavior.

Modifying behaviors is convenient and flexible but can provide the opportunity for malicious persons to modify behavior to their advantage. Special care may be required for plugs 66 from an ICC 5 if they can modify a socket's behavior or be placed in the program flow prior to successful card authentication. For security, the terminal software can specify in accordance with the present invention, a socket control procedure that controls whether or not each individual socket 60 can be modified. Thus, for example, the execution of code downloaded from an ICC 5 can be strictly controlled by the acquirer so that no socket may be plugged from the ICC 5 until all validation routines have been carried out on the card, e.g. examination of an electronic signature.

In accordance with the present invention, socket security includes specifying the socket control procedure to be applied on subsequent attempts to plug a socket 60 (**SETPLUGCONTROL** token). The procedure **PLUGCONTROL** must be written to

return, for a given socket number, whether that socket 60 may now be modified. When a module's socket list is subsequently processed at module load time, or when a socket 60 is plugged programmatically, the VM 20 first executes the user-written **PLUGCONTROL** procedure to determine whether the socket 60 really can be plugged, and retains the existing behavior of the socket 60 if it cannot.

A module that wishes to restrict access to any sockets 60 before another module is loaded for execution may execute a procedure defined by the **SETPLUGCONTROL** token on a pluggable socket (socket zero) with the chosen **PLUGCONTROL** function as a parameter, before loading that module. When the next module, and any other modules loaded for its execution have their socket lists processed, sockets 60 to which access is denied by the user defined **PLUGCONTROL** procedure shall retain their existing behavior. This condition shall not be considered an error. Code that wishes to restrict access to any sockets 60 before further code is executed may execute the procedure defined by the **SETPLUGCONTROL** token with the chosen **PLUGCONTROL** procedure as a parameter, at an appropriate point in program flow. A programmatic request to plug a 60 can determine whether the request was accepted or denied by the call to **SETSOCKET**. Any socket 60 whose behavior was modified, either by the module loading process or dynamically by programmed command, is restored to the behavior it had when the last executable module was loaded for execution, as part of the termination procedure packaged within the procedure defined by the module execute token **(EXECUTEMODULE)**.

As an example of an acquirer-specific function, suppose the basic transaction code includes the phrase 27 **SOCKET LOYALTY** which defines **LOYALTY** and makes it available for later execution. The acquirer's transaction program code further defines code that sets the permission flag for this socket only if the issuer is the same as the acquirer and the transaction amount exceeds a certain minimum. During the transaction there is a command which reads in arbitrary code from the ICC 5. Part of the ICC code could define a **REWARD** routine which updates the user's frequent flier miles, and then attempt to execute the phrase **PLUG REWARD INTO LOYALTY**. This phrase connects the execution of **REWARD** with the execution of **LOYALTY**. If the **LOYALTY** socket permission flag is set according to the above logic, the **SETSOCKET** will take place; otherwise **LOYALTY** will retain its default behavior, likely a no-op. Then when the application code executes its **LOYALTY** function later, it will allow the ICC-

defined **REWARD** only according to the acquirer-defined rules.

Typically, the VM 20 running on a terminal 1 may have a limited number of sockets 60, e.g. 64 sockets, numbered 0 through 63. In its most basic form, a skeleton terminal program could be composed nearly entirely of sockets 60 and basic program flow from socket to socket. The sockets 60 would then be plugged with transaction processing procedures by other modules loaded at application selection time, either from the terminal 1 or from the ICC 5. Sockets 60 occurring in the skeleton program before application selection are assigned a default null behavior by TRS. If a given socket 60 is plugged with a procedure by more than one module, the latest operation simply replaces any earlier ones.

Module loading, handling and execution

Code written to run on the VM 20 (including Terminal Resident Services compiled as token modules) may assume that following power-up the terminal-specific kernel software supporting the VM 20 has performed any necessary terminal-specific power-up initialization, and has started execution of the main processing loop of Terminal Resident Services (TRS) through a module loading process which is described below. If the main processing loop of the TRS is exited, control returns to the terminal-specific layer of software responsible for reloading the TRS and re-entering its main loop. All VM resources are released whenever the TRS exits, except for data in non-volatile databases. Resource releasing occurs when the terminal is powered down, the TRS exits, or the TRS is restarted by the terminal's Operating System (if any). If an updated version of any TRS module has been acquired since the TRS main loop was last entered, all TRS resources *including* data in its non-volatile databases is released when it exits.

Software run on a terminal 1 or an ICC 5 is managed by the VM 20 in the form of one or more *modules*, where each module may contain any of the following categories of information:

- Tokenized code
- Initialized data
- Uninitialized data allocation
- Database definitions
- TLV definitions
- Socket list
- Module interdependencies

Each module is preferably delivered to a terminal 1 in a Module Delivery Format (MDF). The VM 20 maintains a non-volatile *repository* in the read/write non-volatile memory 18 of modules that have been delivered and installed on the terminal 1. Each module in the repository shall be identified by a *module identifier* or *module ID*. Following registration in the module repository, module information is available through a non-volatile module database maintained by the VM 20 and stored in non-volatile memory 18. In accordance with the present invention, the VM 20 protects modules within the repository from modification by any other module because there are no tokens for such an access. Further, the VM 20 makes provision for a new version of a given module to be placed in the repository while a module of the same module ID exists for execution purposes.

There are conceptually two phases in processing a module: firstly it is "loaded," which means it is made accessible and its data, databases, etc. instantiated, and secondly, if it is an executable module, the VM 20 begins processing its tokens starting at its entry point. The execution procedure will be described with reference to the flowchart in Fig. 9.

Firstly, in step 100, the resources are marked and saved. Before execution of a module, the VM 20 marks its state and saves any resources needed so that this state can be restored later. The state includes:

- The position of the extensible memory pointer, the frame pointer, and the frame end pointer.
- The contents of the entire current socket list.
- The TLVs registered in the TLV tag name list.
- Other internal data the VM implementation needs in order to manage the activation and execution of modules.

Next the module is loaded in step 102. The module ID of the module to execute is passed to the *Load Module* subroutine, which will be described later. If the module is loaded without error as determined in step 104, it can be executed and the program progresses to step 108. If an error is determined in step 104, the execution of the module is abandoned and all resources needed for execution of the module are released in step 105. This requires that VM 20 carries out the following actions:

- All volatile memory required to load the module, and any module that it required to be loaded, must be released and cleared to zero. This includes, but is not limited to:
- The space needed for all module's initialized and uninitialized data.
- The space needed for any internal TLV buffers and management data structures de-fined

by these modules.

• The space needed for any internal buffers and management data structures required by databases defined by these modules.

• The TLV name list maintained by the VM for tag lookup must be restored to its state immediately before module execution.

• The contents of the socket list maintained by the VM must be restored to its state immediately before module execution.

• The contents of the frame pointer, frame end pointer, and extensible memory pointer are restored to their values immediately before module execution.

After successful loading of the module, it is determined in step 106 if the module is an executable or a library module. If the latter, no execution of the module takes place and the VM 20 releases all resources in step 110 as described for step 105. If the module is executable the field specifying the entry point of the module is determined.

The VM 20 starts the module by calling the token specified in the entry point field. Then each token is executed in turn in step 108. The module terminates using a **RETURN** token after which all resources are released in step 110.

The process required to load a module, the "Load Module" subroutine, will be described with reference to the flow diagram shown in Fig. 10. If an error is detected during loading of a module, this causes the Load Module subroutine to immediately return *false*. A general error is one such as "out of memory" where there are insufficient resources to provide space for initialized data, uninitialized data, databases, or TLVs; when a duplicate TLV tag is discovered; and so on. Initialized data must be set up before processing the database and TLV sections as these are part of the initialized data section. In step 120 it is determined if the module is already loaded into memory. If it is already loaded, it is not loaded a second time and Load Module immediately succeeds, returning *true*. Next, in step 122, it is determined if the module is in the repository. If not, it cannot be loaded so Load Module subroutine fails, returning *false*. In step 124 is determined how many bytes of data for the module's uninitialized data area 41 are needed and the required amount is reserved. This area 41 is set to all zeroes by the VM 20. Similarly, in step 126, the required number of bytes of data for the module's initialized data area 42 are reserved. Then, the initialized data is copied into this area. In step 18, the TLVs defined in the module to be loaded are added by the VM 20 to its internal name list used for TLV lookup. The root of the TLV data structure is stored. Next, in step 130, the databases defined in the module to be loaded are instantiated by the VM 20. Steps 128 and 130 may

be executed in any order. In step 132, the imported modules for the current module are selected. In step 134, the list of imported modules is traversed, recursively loading each one in turn. If an imported module cannot be loaded for any reason, as determined in step 136, the module that imported the module is also deemed to have failed to load, as it cannot access the imported module's services. In this case, Load Module returns *false*. In step 138 it is determined if a further module is to be imported. If yes, the procedure returns to step 132. After the determination in step 138 that the last imported module has been recursively loaded, the current module has had its resources allocated, loaded, and instantiated without error, so the Load Module plugs the sockets 60 in its list in step 139 and then returns *true* indicating that the module was loaded successfully. Any attempt to plug socket zero must be ignored by the VM 20. If socket zero needs to be plugged, it may be accomplished using the **SETSOCKET** token.

The procedure for plugging the sockets 60 in step 140 will be described with reference to Fig. 11. In step 140 the default behavior for each socket in a loaded module is instantiated. In step 141 it is determined if there is a plug. If no, then the module is executed in step 149. If yes, the first plug is selected in step 142. In 143 it is determined if the security flag of the associated socket is set or not. If not, the socket is plugged in step 146. If yes, the security function specified for the socket is executed. If the security evaluation is positive the socket is plugged in step 146. In step 148 it is determined if the plug is the last plug. If no, the next plug is selected for evaluation. If the security check is negative, it is determined if the plug is the last plug in step 147. If in steps 147 or 148 it is determined that the plug is the last one, the module is executed with the default behavior for all sockets which have not been plugged and for the plugged behavior for those which have been plugged. By this means a secure modification of behavior has been achieved.

Modules that are loaded from an ICC 5 by **LOADCARDMODULE** token must be handled differently than those loaded from the repository in the terminal 1 using the **EXECUTEMODULE** token. The flowchart for **LOADCARDMODULE** is shown in Fig. 12. Before execution of a card module, the VM 20 marks its state and saves any resources needed in step 150 so that this state can be restored later. The state includes:

- The position of the extensible memory pointer, the frame pointer, and the frame end pointer.
- The contents of the entire current socket list.
- The TLVs registered in the TLV tag name list.
- Other internal data the VM implementation needs in order to manage the activation of

card modules.

The module is loaded in step 152 using the Load Module routine described above with reference to Fig. 9; the difference being that the module is on the ICC 5 and is not in the repository and is not already loaded.

If it is determined in step 154 that the card module did not load successfully, all resources are returned in step 155 to the state they had immediately before execution of the **LOADCARDMODULE** token. This requires:

- All volatile memory required to load the module, and any module that it required to be loaded, must be released and cleared to zero. This shall include, but is not limited to:
- The space needed for all module's initialized and uninitialized data.
- The space needed for any internal TLV buffers and management data structures de-fined by these modules.
- The space needed for any internal buffers and management data structures required by databases defined by these modules.
- The TLV name list maintained by the VM for tag lookup must be restored to its state immediately before module execution.
- The contents of the socket list maintained by the VM must be restored to its state immediately before module execution.
- The contents of the frame pointer, frame end pointer, and extensible memory pointer are restored to their values immediately before module execution.

If the card module is loaded successfully, the state saved in the "Mark and save resources" step 150 is simply discarded in step 156. Thus, a card module has been grafted onto a running system. To be useful, a card module must plug sockets otherwise there is no way to execute any code that is present in the card module. Subsequently it is determined in step 158 if the module is an executable module and if so is executed in step 160 as described with reference to steps 106 and 108 of Fig. 9.

The specific embodiments of the invention described above are intended to be illustrative only, and many other variations and modifications may be made thereto in accordance with the principles of the invention. All such embodiments and variations and modifications thereof are considered to be within the scope of the invention as defined in the following claims.

APPENDIX

1. Token Definition

1.1 Overview

EPICode tokens are the instruction set of a two stack virtual machine with an additional frame pointer. The tokens may also be treated as an intermediate language of a compiler. Some implementations of the program translator may actually compile EPICode tokens to machine code.

EPICode tokens are byte tokens, permitting a maximum of 256 tokens. One-byte prefix tokens allow the range of tokens to be extended to a theoretical maximum of 65536 tokens, regarding prefixes as defining pages of 256 tokens each. In fact, a limited range of prefix tokens is defined. Each token value is shown in hexadecimal as a 2-digit hexadecimal code, with its corresponding name.
Tokens without a prefix (single-byte tokens) are referred to as *primary* tokens, whereas those with prefixes (two-byte tokens) are referred to as *secondary* tokens.

The execution of any primary or secondary token that is not defined in the list below will give rise to an `ILLOP` exception.

1.1.1 Forth Functions for EPICode tokens

This section presents an alphabetic concordence list of Forth words used as EPICode tokens. Each line contains, from left to right:

- Definition name, in upper-case, mono-spaced bold-face letters;
- Natural-language pronunciation if it differs from English;
- Special designators, as applicable:
  - A ANS Forth word (including all optional wordsets)
  - C Compiler directive; must be used inside a definition.
  - G Generic Forth word (in common usage, e.g. Forth Interest Group, but not in ANS Forth).
  - H Host (compiler) word, which may or may not contribute tokens to the target.
- Equivalent EPICode token(s).

Word	Pronunciation	Codes	EPICODE Tokens
-	minus	A	SUB
!	store	A	STORE
#	number-sign	A	NMBR
#>	number-sign-greater	A	NMBRGT
#S	number-sign-s	A	NMBRS
*	star	A	MUL
/	slash	A	DIV
:	colon	A,H	PROC
;	semi-colon	A,C,H	ENDPROC
?DO	question-do	A,C,H	RQDO <+addr>
?DUP	question-dupe	A	QDUP
?THROW		G	QTHROW
@	fetch	A	FETCH
[']	bracket-tick	A,C,H	LITC <+addr>
[CHAR]	bracket-char	A,C,H	PLIT <n>
+	plus	A	ADD
+!	plus-store	A	INCR
+LOOP	plus-loop	A,C	RPLUSLOOP <+addr>
<	less-than	A	CMPLT
<#	less-number-sign	A	LTNMBR
<>	not-equals	A	CMPNE

APPENDIX

=	equals	A	CMPEQ
>	greater-than	A	CMPGT
>BODY	to-body	A,H	WLIT <+addr>
>NUMBER	to-number	A	TONUMBER
>R	to-r	A,C	TOR
0<	zero-less	A	SETLT
0<>	zero-not-equals	A	SETNE
0=	zero-equals	A	SETEQ
0>	zero-greater	A	SETGT
1-	one-minus	A	SSUBLIT 1
1+	one-plus	A	SADDLIT 1
2!	two-store	A	VSTORE
2*	two-star	A	SHL
2/	two-slash	A	SHR
2@	two-fetch	A	VFETCH
2>R	two-to-r	A,C	TWOTOR
2DROP	two-drop	A	TWODROP
2DUP	two-dupe	A	TWODUP
2OVER	two-over	A	TWOOVER
2R@	two-r-fetch	A,C	TWORFETCH
2R>	two-r-from	A,C	TWORFROM
2ROT	two-rote	A	TWOROT
2SWAP	two-swap	A	TWOSWAP
2VARIABLE	two-variable	A,H	LITU <+addr>
ABS	abs	A	ABS
AGAIN		A,C,H	BRA <+addr>
AND		A	AND
BASE		A	USERVAR 1
BUFFER:	buffer-colon	G,H	LITU <+addr>
C!	c-store	A	BSTORE
C@	c-fetch	A	BFETCHU
C+!	c-plus-store	G	BINCR
CATCH		A	CATCH
CELL		G	FOUR
CELL+	cell-plus	A	SADDLIT 4
CELLS		A	SMULLIT 4
CHAR	char	A	PLIT <n>
CHAR+	char-plus	A	SADDLIT 1
CHARS	chars	A	NOOP
COMPARE		A	BCMP
CONSTANT		A,H	LIT <x>
D+	d-plus	A	VADD
DEPTH		A	DEPTH
DNEGATE	d-negate	A	VNEGATE
DO		A,C,H	RDO

APPENDIX

DROP		A	DROP
DUP		A	DUP
ELSE		A,C,H	BRA <+addr>
ENDCASE	end-case	A,C,H	DROP
ENDOF	end-of	A,C,H	BRA <+addr>
EXECUTE		A	ICALL
EXIT		A,C	RETURN
FILL		A	BFILL
GET_MSECS		G	GETMS
HOLD		A	HOLD
I		A,C	RI
IF		A,C,H	BZ <+addr>
INVERT		A	INVERT
J		A,C	RJ
LEAVE		A,C	RLEAVE <+addr>
LITERAL		A,C,H	LIT <n>
LOCALS|	locals-bar	A,C,H	<method> <addr>
LOOP		A,C,H	RLOOP <+addr>
LSHIFT	l-shift	A	SHLN
M*	m-star	A	MMUL
M/MOD	m-slash-mod	G	MSLMOD
MAX		A	MAX
MIN		A	MIN
MOD		A	MOD
MOVE		A	BMOVE
MS		A	MS
NEGATE		A	NEGATE
NIP		A	NIP
NOT			SETEQ
OF		A,C,H	ROF <+addr>
OR		A	OR
OVER		A	OVER
PICK		A	PICK
PLUG		H	LITC <+addr>
RECURSE		A,H	CALL
REPEAT		A,C,H	BRA <+addr>
-ROT	minus-rot	G	ROTB
R@	r-fetch	A,C	RFETCH
R>	r-from	A,C	RFROM
ROT		A	ROT
RSHIFT	r-shift	A	SHRNU
SIGN		A	SIGN
SOCKET		H	DOSOCKET <n>
SWAP		A	SWAP
THROW		A	THROW

APPENDIX

TIME&DATE	time-and-date	A	GETTIME
-TRAILING	minus-trailing	A	MINUSTRAILING
TUCK		A	TUCK
U<	u-less-than	A	CMPLTU
U<=	u-less-than-or-equal	G	CMPLEU
U>	u-greater-than	A	CMPGTU
U>=	u-greater-than-or-equal	G	CMPGEU
UM*	u-m-star	A	MMULU
UM/MOD	u-m-slash-mod	A	MSLMODU
UMOD	u-mod	G	MODU
UNTIL		A,C,H	BZ <+addr>
USER		G	USERVAR <n>
VALUE		A,H	LITD <+addr> <method>
VARIABLE		A,H	LITU <n>
WHILE		A,C,H	BZ <+addr>
WITHIN		A	WITHIN
XOR		A	XOR

1.2 Conventions

1.2.1 Number Formats

Numbers larger than one byte are transmitted in token programs in "big-endian" 2's complement form, most significant byte first. Within an EPICode program, numbers should always be accessed by operators of the correct format, in order to allow programs to store numbers in the form most suited to the underlying architecture.

Multiple-precision data types are held on the stack with the most significant cell topmost. In memory, these data types are held with the most significant cell at the lowest-addressed cell within the multi-cell type.

1.2.2 Control Structures and Offsets

Control structures are formed by a control token (BRA, RLOOP, etc.) followed by a signed four-byte, two-byte or single byte offset. The offset following the control token is added to the Token Pointer (TP) after the offset has been fetched. Thus, if a branch token is at *addr*, the destination address is *addr+2+offset* for a 1-byte offset (SBRA), *addr+3+offset* for a 2-byte offset (BRA), and at *addr+5+offset* for a 4-byte offset (EBRA).

Tokens taking four-byte offsets are available only to terminal-specific code on virtual machine implementations that support a 32-bit linear address space for code.

1.2.3 Addresses

User-defined procedures are defined by their addresses within the EPICode program. If the tokens are translated or native-code compiled for larger processors, the token space address will not correspond to the actual address of the code.

1.3 Data Typing

Most tokens operate on quantities with a data size and a signed or unsigned interpretation determined by the token, but instructions which access memory in the frame store can take a data type override determined by a prefix token. A set of byte codes, shown in Table 1, is reserved for such prefix tokens, but only SBYTE and UBYTE are currently used.

Data types that require less than one cell (1 byte) are fetched from memory by using a byte operator or an

operator with an override prefix. If a signed data type is implied or specified, the data is sign-extended to cell width. If an unsigned data type is implied or specified, the data is zero-extended.

Prefix	Abbr.	Description	Size
SBYTE	SB	Signed Byte	1 byte
UBYTE	UB	Unsigned Byte	1 byte
SLONG	SL	Signed Long	4 bytes
ULONG	UL	Unsigned Long	4 bytes
SVLONG	SV	Signed Vlong	8 bytes
UVLONG	UV	Unsigned Vlong	8 bytes

Table 1: Data type prefixes

1.4 Arithmetic

Addition and subtraction operations that overflow the size specified for the returned result will return that result modulo [maximum unsigned value accommodated in that size]+1.

Store operations whose destination storage is smaller than the size of the value passed will store the value truncated to the width of the destination.

Division operations are symmetric; that is, rounding is always towards zero regardless of sign.

1.5 Primary Tokens

Tokens are split into several logical sets for the sake of convenience and are shown in separate sections below. All token values are in hexadecimal.

Data type prefixes applicable to the tokens are listed explicitly, using the abbreviations given in Table 1. Any primary token which is prefixed by a token not in its prefix list is invalid, and the execution of such a token causes an ILLOP exception to be thrown. The default type for the token is italicised, and is always listed first. The default data type prefix is redundant, and is invalid if used to prefix a token, as above.

1.5.1 Operations Set

00 NOOP

(—)

No action is taken.

04 BFETCHS

(addr — num)

Fetch an 8-bit byte from the given address, sign extending it.

08 LIT

(— x)

Return the cell that follows in-line as data.

09 LITC

(— addr)

Return the cell that follows in-line as a literal that is an address in the code image. The value of the literal may be relocated in that image by the program loader.

0A LITD

(— addr)

Return the cell that follows in-line as a literal that is an address in initialized data space. The value of the literal may be relocated in that image by the program loader.

0B LITU

(— addr)

Return the cell that follows in-line as a literal that is an address in uninitialized data space. The value of the literal may be relocated in that image by the program loader.

0C PLIT

(— u)

Return the byte that follows in-line. The byte is zero-extended to 32 bits.

0D NLIT

(— num)

Return the byte that follows in-line, zero-extended to 32 bits and then negated.

0E HLIT

(— u)

Return the 2-byte value that follows in-line. The value is zero-extended to 32 bits.

10 HLITC

(— addr)

Return the address resulting from adding the unsigned 2-byte value that follows in-line to the base address of the code image. The value is zero-extended to 32 bits.

11 SLITD

(— a-addr)

Return the address resulting from adding the unsigned byte that follows in-line interpreted as a positive offset in cells to the base address of initialized data. The byte is zero-extended to 32 bits and multiplied by 4 to give an offset in bytes.

12 HLITD

(— addr)

Return the address resulting from adding the 2-byte value that follows in-line to the base address of initialized data. The value is sign-extended to 32 bits.

13 SLITU

(— addr)

Return the address resulting from adding the unsigned byte that follows in-line, interpreted as a positive offset in cells, to the base address of uninitialized data. The byte is zero-extended to 32 bits and multiplied by 4 to give an offset in bytes.

14 HLITU

(— addr)

Return the address resulting from adding the 2-byte value that follows in-line to the base of uninitialized data. The value is sign-extended to 32 bits.

15 ADDLIT

(x_1 — x_2)

Add the data in the cell that follows in-line to x_1, yielding x_2.

16 SADDLIT

(x_1 — x_2)

Add a literal from the signed one-byte in-line value to x_1, yielding x_2.

19 SUBLIT

(x_1 — x_2)

Subtract the data in the cell that follows in-line from x_1, yielding x_2.

1A SSUBLIT

(x_1 — x_2-t)

Subtract the signed one-byte in-line value from x_1, yielding x_2.

1C VSUBLIT

(d — d-lit)

Subtract the signed eight-byte in-line value from the double number *d*.

1D SMULLIT

*(num — num*lit)*

Multiply *num* by the signed one-byte literal that follows inline.

1E SDIVLIT

(num — num/lit)

Divide *num* by the signed one-byte literal that follows inline.

21 DIVU

(u_1 u — u_3)

Divide u_1 by u_2 (unsigned) giving u_3.

APPENDIX

3A **SHRU**

(u — u>>1)

Logical shift *u* right one bit, inserting a zero bit.

N.B. The **SETxx** operators perform a comparison with zero, the flag being set according to the results of this comparison.

42 **SETGE**

(num — flag)

Return TRUE if *num* ≥ 0 (signed).

45 **SETLE**

(num — flag)

Return TRUE if num ≤ 0 (signed).

48 **CMPGEU**

(u_1 u_2 — flag)

Compare the unsigned values u_1 and u_2 returning TRUE if $u_1 \geq u_2$.

4C **CMPGE**

(num_1 num_2 — flag)

Compare the signed values num_1 and num_2 returning TRUE if $num_1 \geq num_2$.

4F **CMPLE**

(num_1 num_2 — flag)

Compare the signed values num_1 and num_2 returning TRUE if $num_1 \leq num_2$.

The following tokens provide access to the frame store.

50..53 **PFRFETCH2 . . . PFRFETCH5**

(— num)

Short form equivalents for **SFRFETCH** *n* (q.v.), where *n* is 2..5.

Possible data type overrides include: *SL*, SB, UB

54..5F **TFRFETCH12 . . . TFRFETCH1**

(— num)

Short form equivalents for **SFRFETCH** *n* (q.v.), where *n* is -12..-1.

Possible data type overrides include: *SL*, SB, UB

60..63 **PFRSTORE2 . . . PFRSTORE5**

(num —)

Short form equivalents for **SFRSTORE** *n* (q.v.), where *n* is 2..5.

Possible data type overrides include: *SL*, SB

64..6F **TFRSTORE12 . . . TFRSTORE1**

(num —)

Short form equivalents for **SFRSTORE** *n* (q.v.), where *n* is -12..-1.

Possible data type overrides include: *SL*, SB

70 **SFRFETCH**

(— num)

Fetch the value (by default, cell) *num* at the signed in-line one-byte offset from the frame pointer. The offset is interpreted as a cell index (that is, it is multiplied by 4 to give a byte-addressed offset) for the default data type, and as a byte index for a byte-sized data override. Note that **SFRFETCH 0** and **SFRFETCH 1** return internal frame management data with no meaning to the calling program, and so do not constitute valid references into the frame. Thus parameters start at **SFRFETCH 2** and temporary variables start at **SFRFETCH -1**, since frame stacks grow downwards in memory.

Possible data type overrides include: *SL*, SB, UB

71 **SFRSTORE**

(num —)

Store the value (by default, cell) *num* in the argument at the signed in-line one-byte offset

from the frame pointer. The offset is supplied as an in-line value which is treated as a cell index (that is, it is multiplied by 4 to give a byte-addressed offset) for the default data type, and as a byte index for a byte-sized data override. See SFRFETCH for further details.
Possible data type overrides include: *SL*, SB

72 FRFETCH

(— num)
Fetch the value *num* at the signed offset from the frame pointer. The offset is provided by a two-byte in-line value. See the description of SFRFETCH for more details.
Possible data type overrides include: *SL*, SB, UB

73 FRSTORE

(num —)
Store the value *num* in the argument at the signed offset from the frame pointer. The offset is provided by a two-byte in-line value. See SFRSTORE for further details.
Possible data type overrides include: *SL*, SB

74 SFRADDR

(— addr)
Return the address in the frame at the signed offset from the frame pointer. The offset is provided by a one-byte in-line value which is multiplied by 4 to give a byte offset for the default data type, and used directly as a byte index for a byte-sized data override.
Possible data type overrides include: *SL*, SB

75 FRADDR

(— addr)
Return the address in the frame at the signed cell offset from the frame pointer. The offset is provided by a two-byte in-line value which is multiplied by 4 to give a byte offset for the default data type, and used directly as a byte index for a byte-sized data override.

For tokens providing support for Forth standard number conversion functions: the NMBR in the token names is pronounced "number". Tokens LTNMBR, NMBRS and TONUMBER employ the user variable BASE as the conversion number base.

8C UNDER F

(x_1 x_2 — x_1 x_1 x_2)
Duplicate the second item on the stack.

9C ZERO

(— 0)
Leave the value 0 on the stack.

9D ONE

(— 1)
Leave the value 1 on the stack.

9E MINUSONE

(— -1)
Leave the value -1 on the stack.

A0 INDEX

($addr_1$ num — $addr_2$)
Multiply *num* by 4 and add to $addr_1$ to give $addr_2$.

A2 EDOCREATE

(— a-addr)
Return the address in data space whose offset follows in the in-line cell immediately after this token, and perform a subroutine return. This token is used to identify a data area by calling a procedure corresponding to it, permitting the creation of position-independent data tables.

A3 EDOCLASS

(— a-addr)
Branch to the code space address whose offset is held in the in-line cell that follows, after

pushing onto the data stack the address resulting from adding the unsigned offset that follows in the cell next in-line (i.e. after the code offset) to the base address of initialized data space. This token is used to identify a data structure in program memory and to transfer control to the routine that processes it, providing the basis of a simple class mechanism.

A4 DOCREATE

(— a-addr)

Return the address in data space whose offset follows in the in-line two-byte value immediately after this token, and perform a subroutine return. This token is used to identify a data offset by calling a procedure corresponding to it, permitting the creation of position-independent data tables.

A5 DOCLASS

(— a-addr)

Branch to the code space address whose offset is held in the in-line cell that follows, after pushing onto the data stack the address resulting from adding the unsigned offset that follows in the two bytes next in-line (i.e. after the code offset) to the base address of initialized data space... This token is used to identify a data structure in program memory and to transfer control to the routine that processes it, providing the basis of a simple class mechanism.

A6 ECALL

(—)

Followed by an in-line cell, calls the procedure using this cell as a signed byte offset into code space.

A7 SCALL

(—)

Followed by one in-line byte, calls the procedure using this byte as a signed byte offset into code space.

A8 CALL

(—)

Followed by an in-line two-byte offset, calls the procedure using this value as a signed byte offset into code space.

AB SMAKEFRAME

(x_{params} . . . x_1 —)

Followed by two unsigned one-byte literals containing first *params*, the number of cells forming the procedure parameters, and then *temps*, the number of cells of temporary variables. Allocates *params+temps+2* cells, and then sets the current Frame Pointer to point to the new frame. This token allows procedure parameters and temporary variables to be accessed by FRFETCH and FRSTORE.

The virtual machine is permitted to build frames on the return stack, so use of frames is constrained by the rules that apply to return stack usage in general. Procedure parameters will be moved from the data stack into the frame by SMAKEFRAME so that they can be accessed by FRFETCH and FRSTORE.

If it is not possible to build a frame of the requested size, a FRAME_STACK_ERROR will be thrown.

AC MAKEFRAME

(x_{params} . . . x_1 —)

Followed by two unsigned two-byte literals containing first *params*, the number of cells forming the procedure parameters, and then *temps*, the number of cells of temporary variables. See SMAKEFRAME for further details.

AD RELFRAME

(—)

Restore the frame pointer to its previous value and release the current frame.

APPENDIX

1.5.2 Branch Set

These tokens include the usual stack-machine branch operators, plus the runtimes for the Forth words DO ?DO LOOP +LOOP LEAVE I and J.

AF EBRA
(—)
Branch always. Four-byte inline offset

B0 EBZ
(num —)
Branch if *num* = 0. Four-byte inline offset.

B1 EBNZ
(num —)
Branch if *num* ≠ 0. Four-byte inline offset.

B2 SBRA
(—)
Short branch. Signed byte inline offset.

B3 SBZ
(num —)
Short branch if *num* = 0. Signed byte inline offset.

B4 SBNZ
(num —)
S hort branch if *num* ≠ 0. Signed byte inline offset

B5 BRA
(—)
Unconditional branch. Signed two-byte inline offset.

B7 BNZ
(num —)
Branch if *num* ≠ 0. Signed two-byte inline offset.

1.5.3 Data Type and Code Page Overrides

This group allows the limitations of 8-bit tokens to be overcome. Note that their stack action depends on the following token. The pair of tokens is referred to as a *secondary* token.
The extension tokens for data types occupy tokens C0 to CF. Unused tokens in this range are reserved for future use when additional data type prefixes are required.

C1 SBYTE
(—)
Signed Byte.

C2 UBYTE
(—)
Unsigned Byte.

C5 SLONG
(—)
Signed Long, 32 bits

C6 ULONG
(—)
Unsigned Long, 32 bits

1.5.4 Socket Handling Tokens

D2 DOSOCKET
(—)
Followed by an in-line byte (0..63) which specifies the function number required. The stack effect is defined by the function attached to the socket.

D3 IDOSOCKET

(u —)

Executes the socket function whose socket number (0..63) is specified by *u*. The lower-level stack effect is defined by the function attached to the socket. ANS Forth exception 24 (invalid numeric argument) will be thrown if *u* is greater than 63.

1.5.5 Control Set

E6 IMCALL

(—)

Execute the function from the module whose module number (0-255) is given in the next byte in-line, and whose function number (0-255) is given in the following byte in-line. Stack effects are dependent on the function called.

E7 CLASSPROC

(—)

During loading CLASSPROC marks the entry to class handling code. Used for compilation assistance and may be implemented as a NOOP.

F9 SYSFUNC

(—)

A page expansion token treated as the first byte of a secondary token. Calls the routine specified by the following in-line byte. The supported secondary token set is defined in Section 1.7. The stack effect is defined by the specified routine.

1.6 Sockets

The first eight secondary socket tokens are reserved for socket management functions, and defined management functions are described below. Remaining sockets (D2 08 to D2 3F) are for application use.

F9 91 SETSOCKET

(xp u — flag)

Set the execution pointer *xp* to be the handler of socket function *u*, which will cause *xp* to be executed by a subsequent execution of DOSOCKET <u>. Before the execution pointer is set, the procedure installed by SETPLUGCONTROL is run to determine whether the socket may be plugged with this new *xp*. *flag* is the value returned by this procedure. SETSOCKET will only set the pointer if *flag* is FALSE, otherwise the pointer is discarded. Exception -24 (invalid numeric argument) will be thrown if *u* is greater than 63.

D2 00 SETPLUGCONTROL

(xp —)

Stores the execution pointer *xp* of a user-written procedure that will be run by SETSOCKET to determine whether the socket can be plugged.

The action of this procedure (which is here referred to as PLUGCONTROL for purposes of illustration) must be:

(u — flag)

where *u* is the socket number and *flag* is returned FALSE if the socket can be plugged, or TRUE if it cannot. In addition, the PLUGCONTROL procedure must raise an exception -24 (invalid numeric argument) for values of u outside the range 0-63.

A default action of PLUGCONTROL is installed by the Virtual Machine to return FALSE for all values of *u*, enabling all sockets to be plugged.

D2 03 OSCALLBACK

(dev fn num_1 num_2 — ior)

Calls an operating system routine with the parameters: *dev* selects the requested I/O device for function *fn* with num_1 32 bit parameters contained in array num_2 , returning *ior* which is implementation dependent. Note that num_2 is on the top of the stack. num_1 and num_2 correspond to *arvc* and *argv*, respectively, in C usage.

Note that this socket is implementation dependent, and is provided so that terminal specific programs (TRS) written using EPICode can have terminal dependent I/O. If the specified

function is unsupported, exception -21 (unsupported operation) is thrown.

D2 04 EPICALLBACK

(dev fn num$_1$ num$_2$ — ior)

A socket for an EPICode routine which may be called by the underlying operating system. The four parameters are 32-bit values intended for use as follows: *dev* selects the requested I/O device for function *fn* with *num$_1$* 32-bit parameters contained in the array *num$_2$*, returning *ior* whose meaning is implementation dependent. *num$_1$* and *num$_2$* correspond to *arvc* and *argv*, respectively, in C usage.

Note that this socket is implementation-dependent, and is provided so that terminal specific programs (TRS) written using EPICode can provide callback routines for the operating system. If the specified function is unsupported, exception -21 (unsupported operation) is thrown.

1.7 SYSFUNC I/O Set

This set defines the functions available via the SYSFUNC token, which acts as a generalised interface to underlying operating system routines.

1.7.1 Device Access

Each device is assigned a unique device number. Status *ior* codes are device dependent, except that an *ior* code of 0 always indicates success.

F9 00 DKEY

(dev — echar)

Read a character from input device *dev*.

F9 01 DKEYTEST

(dev — flag)

Return TRUE if a character is ready to be read from input device *dev*.

F9 02 DEMIT

(char dev —)

Transmit *char* to output device *dev*.

F9 03 BEEP

(u dev —)

Request the output device *dev* to generate a sound for duration *u* milliseconds. This function may suspend processing for the specified duration.

F9 04 DREAD

(addr len dev — ior)

Read a string from input device *dev*, returning a device-dependent *ior*. The string returned contains only the lower order byte of characters read from a keyboard device.

F9 05 DWRITE

(addr len dev — ior)

Write a string to output device *dev*, returning a device-dependent *ior*.

F9 06 DSTATUS

(dev — ior)

Return the status *ior* of the resource associated with device *dev*, where in the general case "ready" and "serviceable" is indicated by 0 and "not ready" is indicated by any other value. A specific device may return non-zero values that have significance for that device. If the device has been selected by a previous execution of the OUTPUT token, the DSTATUS should return "not ready" until execution of the function passed to OUTPUT completes.

F9 07 DIOCTL

(dev fn num a-addr — ior)

Perform IOCTL function *fn* for channel *dev* with *num* cell-sized arguments in the array at *a-addr*.

F9 08 OUTPUT

(xp dev — ior)
Execute the procedure whose execution pointer is given by *xp* with output being directed to device *dev*. On return from OUTPUT the current output device (see GETOP) is unaffected. *ior* is returned as zero if the procedure is executable. All exceptions arising from the execution of *xp* are caught by the Virtual Machine and cause immediate termination of OUTPUT.

F9 09 DWRITESTRING
(dev —)
This token is followed by a string of characters, stored in the token stream as a count byte followed by that many bytes. The DWRITESTRING token writes the characters to the selected device *dev*. Execution continues immediately after the last character.

F9 0A GETOP
(— dev)
Returns the device *dev* last selected by SETOP or during execution of a function passed to OUTPUT. Used to find the default device for device oriented I/O operations. This function allows functions dependent on a current device to be implemented easily.

F9 0B SETOP
(dev —)
Used to set the default device *dev* returned by GETOP for device oriented I/O operations. This function allows functions dependent on a current device to be implemented easily.

F9 0C FORMFEED
(dev —)
Executes a device-dependent "new form" action such as "clear screen" (terminal display) or "page throw" (printer) on device *dev*.

F9 0D CR
(dev —)
Executes a device-dependent "new line" action on device *dev*.

F9 0E SETXY
(num_1 num_2 dev —)
Executes a device-dependent "set absolute position" action on device *dev* using num_1 as t he x-ordinate and num_2 as the y-ordinate.

1.7.2 Time handling

Standard Forth words.

1.7.3 Language and Message Handling

Tokens in this group provide a mechanism for handling language and message selection and display.

F9 20 CHOOSELANG
(addr — flag)
Select the language whose ISO 639 language code is given by the two characters at *addr*. If *flag* is TRUE, the language was found and is now the current language. Otherwise, the calling program should select another language. At least one language (the terminal's native language) will always be available.

F9 21 CODEPAGE
(num — flag)
Attempts to select the resident code page *num* Code pages are numbered according to ISO 8859 (0 = common character set, 1 = Latin 1, etc.). *flag* is TRUE if the code page has been selected.

F9 22 LOADPAGE
(addr — flag)
Install the code page at *addr* in the terminal (this page will usually be found in the card). flag indicates a successful upload of the page. Page installation may be done when a new message table has been loaded from the ICC that requires a code page that is not available

on the terminal.

F9 23 INITMESSAGES

(—)

This function erases private issuer messages, numbered from C0 to FF (hex) and any messages installed by LOADMESSAGES. This function should be called after each user session.

F9 24 LOADMESSAGES

(c-addr —)

Install a message table in an appropriate place in the transient messages data base. *c-addr* gives the location of the message table definition, including the page code to use for messages, the two-letter language code in accordance with ISO 639, and the messages to be installed.

F9 25 GETMESSAGE

(num — c-addr len)

Returns string parameters for message *num*. Trailing spaces are removed from the length *len* of the string.

F9 27 UPDATEMESSAGES

(addr len —)

Install a message table into the resident language tables. If a language with the same code is already present, it will be replaced; otherwise, the new language will be added. If there is not sufficient space for the new language, a THROW will be issued with code 22 (TOO_MANY_LANGUAGES).

addr gives the location of the TLV containing the message table definition, including the page code to use for messages, the two-letter language code in accordance with ISO 639, and the messages to be installed.

F9 28 MESSAGESIZE

(— len)

Returns the standard length of messages for this terminal.

F9 29 TYPEMESSAGE

(addr len —)

Displays the given string on the message line of the terminal.

1.7.4 ICC Card Handling

Tokens in this group provide a mechanism for handling Integrated Circuit Card readers.

F9 30 INITCARD

(num — ior)

Selects ICC reader *num*, where *num* is 0 or 1.

F9 31 CARD

($c\text{-}addr_1$ len_1 $c\text{-}addr_2$ len_2 — $c\text{-}addr_2$ len_3)

Send the data in the buffer $c\text{-}addr_1$ len_1 to the card, and receive data at $c\text{-}addr_2$ len_2. The returned len_3 gives the actual length of the string received.

The buffer $c\text{-}addr_1$ len_1 must contain:

Four-byte standard ISO header (Class, Instruction, P1, P2)

Optional data (*length* followed by *length* bytes, where *length* may be 0-255).

The buffer $c\text{-}addr_2$ len_2 must provide adequate space for the answer from the card plus two status bytes containing SW1 and SW2.

Error handling is done internally within CARD.

F9 32 CARDON

($c\text{-}addr$ len_1 — $c\text{-}addr$ len_2 ior)

Apply power to ICC and execute the card reset function. $c\text{-}addr$ len_1 provides a buffer into which the Answer to Reset will be placed; len_2 is the actual length of the string returned.

F9 33 CARDOFF

(—)

Power off ICC. Executed when all transactions are complete.

F9 34 CARDABSENT

(— flag)

Return TRUE if an ICC card is not present in the reader, FALSE otherwise.

1.7.5 Magnetic Stripe Handling

Tokens in this group provide a mechanism for handling Magnetic Stripe devices

F9 38 FROMMAG

(c-addr len_1 num — c-addr len_2 ior)

Read one or more ISO magstripes. The operation can be interrupted by user CANCEL key or by a time-out. *num* is the ISO identifier of the magstripe track(s) to read[1] , *c-addr* is the destination address for the string, and len_1 is its maximum length (at least 78 bytes for ISO1, 41 bytes for ISO2 and 108 for ISO3, or sums of these for reading multiple magstripes). On return, len_2 gives the actual length of the string read.

F9 39 TOMAG

(c-addr len num — ior)

Write one ISO magstripe. The data is in the buffer *c-addr len* and will be written to stripe *num* (1-3). Operation can be interrupted by user CANCEL key or by time-out.

1.7.6 Modem Handling

Tokens in this group provide a mechanism for handling the modem device.

F9 40 MODEMCALL

(num_1 num_2 num_3 num_4 num_5 c-addr len —ior)

Calls a number using an internal terminal modem.

num_1 and num_2 indicate the speed of the input and output line to use (from 75 to 19200 baud). The actual speeds supported are implementation-defined.

num_3 indicates parity (0 = none, 1 = odd, 2 = even).

num_4 indicates number of bits to use (7 or 8).

num_5 indicates number of stop bits used for transmission (1 or 2 bits).

c-addr len is a string containing the phone number to call. ',' can be included for dial tone waiting. If the first character of this string is 'P', pulse dialling is used instead of default tone dialling.

F9 41 MODEMHANGUP

(— ior)

This function is used to end the current modem session.

F9 42 TOMODEM

(c-addr len — ior)

Transmit the string at *c-addr len* on an established modem session.

F9 43 FROMMODEM

(c-addr len_1 — c-addr len_2 ior)

Receives a string from the modem. *c-addr* is the destination address for the string, and len_1 is its maximum length. On return, len_2 gives the actual length of the string read. If no characters are received for a specified period, a time-out occurs.

F9 44 MODEMBREAK

(— ior)

This function sends a break on the connected modem session.

1.7.7 Blacklist Management

Tokens in this group provide a mechanism for handling the blacklist file.

F9 48 INITBLACKLIST

(—)

1

This function initializes the black list to an empty state.

F9 49 BLACKLISTINSERT

(c-addr len — flag)

This function inserts an entry at *c-addr len* in the list, which is maintained in sorted order. This function must be used when updating the list.

The returned *flag* is FALSE if the insertion was successful (the entry was not found in the existing list and the list was not full).

F9 4A INBLACKLIST

($c\text{-}addr_1$ len_1 — $c\text{-}addr_2$ len_2 flag)

This function attempts to find a key $c\text{-}addr_1$ len_1 in the list.

$c\text{-}addr_2$ len_2 contain the result of the search (including remaining bytes from the selected entry and possibly some other information bytes), if the key was found.

The returned *flag* is FALSE if the number was found.

F9 4B BLACKLISTDELETE

(c-addr len — flag)

This function deletes an entry from the list, where *c-addr len* is the key for the entry to delete. It can be up to 18 bytes long.

The returned *flag* is FALSE if the deletion was successful (the entry was found).

1.7.8 Support for Security Algorithms

Tokens in this group provide support for initializing and using security services.

F9 50 INITSECALGO

(c-addr len num — flag)

c-addr is the address of initialization buffer, and *len* is its length. The input parameter(s) for each algorithm may differ, though a key should usually be passed to this initialization

flag is FALSE if initialization occurred successfully.

F9 51 SECALGO

($c\text{-}addr_1$ len $c\text{-}addr_2$ num — flag)

Were $c\text{-}addr_1$ is the input data buffer for computation, and *len* its length. $c\text{-}addr_2$ is the output buffer for storage of the result.

flag is FALSE if computation occurred successfully.

Terminal Services

1.7.9 Terminal Services

F9 58 POWERLESS

(— flag)

Returns FALSE if there is enough power to complete the current transaction.

1.7.10 Database Services

The following tokens provide a mechanism for handling databases.

F9 61 DBMAKECURRENT

(a-addr —)

Make the database whose DPB is at *a-addr* the current database.

F9 62 DBSIZE

(— len)

Returns the size of the record buffer that provides the window onto the current record of the current database.

F9 63 DBFETCHCELL

(num_1 — num_2)

Returns the 32-bit value num_2 from the cell at the cell-aligned byte offset num_1 in the current record of the current database.

F9 64 DBFETCHBYTE

(num — char)

Returns the one-byte value *char* from the byte offset *num* in the current record of the

current database.

F9 65 DBFETCHSTRING

(num len — addr len)

Returns the string parameters *addr* and *len* of the byte sequence at offset *num* and with length *len* in the current record of the current database.

F9 66 DBSTORECELL

(num_1 num_2 —)

Stores the 32-bit value *num_1* to the cell at the cell-aligned offset *num_2* in the current record of the current database and updates the database record.

F9 67 DBSTOREBYTE

(char num —)

Stores the 1-byte value *char* to the byte at offset *num* in the current record of the current database and updates the database record.

F9 68 DBSTORESTRING

(addr len_1 num len_2—)

Stores at most *len_2* bytes of the byte sequence at *addr* to offset *num* in the current record of the current database and updates the database record. If *len_1* is less than *len_2* then the destination in the database record buffer is space filled to *len_2*.

F9 69 DBINITIALIZE

(—)

Initializes the current database to all zeros, and sets "current" and "available" record numbers of the database (see DBRECNUM and DBAVAIL) to 0.

F9 6A DBRECNUM

(— u)

Returns the current record number.

F9 6B DBCAPACITY

(— u)

Returns the total number of records that the current database can hold.

F9 6C DBAVAIL

(— num)

Returns the record number of the next available record in the current file.

F9 6D DBADDREC

(—)

Add a record at the end of the current database, at the record number given by DBAVAIL.

F9 6F DBSELECT

(num —)

Select record *num* in the currently selected database.

F9 70 DBMATCHBYKEY

(addr len — flag)

Search the current database for a match on the key field against the string specified by *addr* and *len*. *Len* may be shorter than the defined length of the key field for this structure, with the remaining characters being compared to blank (ASCII 20h) characters. If the match is successful, the matching record becomes current and *flag* is FALSE.

This token may only be used with an Ordered database.

F9 71 DBADDBYKEY

(addr len — flag)

Search the current database for a match on the key field against the string specified by *addr* and *len*. *Len* may be shorter than the defined length of the key field for this structure, with the remaining characters being compared to blank (ASCII 20h) characters. If the match is successful, the matching record becomes current and *flag* is TRUE. If the match is not successful, a new record is inserted at the correct position in the database and the flag is FALSE. This new record will be initialized except for its key field which will contain the given key.

This token may only be used with an Ordered database.

F9 72 DBDELBYKEY F
(addr len — flag)
Search the current database for a match on the key field against the string specified by *addr* and *len*. *Len* may be shorter than the defined length of the key field for this structure, with the remaining characters being compared to blank (ASCII 20h) characters. If the match is successful, the matching record is deleted and *flag* is FALSE. The deletion action closes up any potential "hole" in a physical implementation by taking appropriate action to physically reposition or relink the records in a pre-initialized database. This token may only be used with an Ordered database.
F9 73 DBSAVECONTEXT
(—)
Causes the server to stack the current context information, including the current database, the current record number and any ancillary information. The server is entitled to use the Virtual Machine's return stack to save context information, and client software *must* therefore observe the general rules that apply for return stack usage.
F9 74 DBRESTORECONTEXT
(—)
Causes the server to restore the most recently saved context information (see DBSAVECONTEXT). The server is entitled to use the Virtual Machine's return stack to save context information, and client software *must* therefore observe the general rules that apply for return stack usage.

1.8 TLV Management

The tokens described in this section provide TLV management and access function.

1.8.1 String Processing Suport

F9 78 PLUSSTRING
(c-addr$_1$ len$_1$ c-addr$_2$ len$_2$ — c-addr$_2$ len$_3$)
Store the string at *c-addr$_1$* for *len$_1$* bytes at the end of the string at *c-addr$_2$* for *len$_2$* bytes. Return the beginning of the destination string (*c-addr$_2$*) and the sum of the two lengths (*len$_3$*). There must be room at the end of the destination string to hold both strings.
F9 79 CPLUSSTRING
(char c-addr len — c-addr len+1)
Store the character *char* at the end of the string at *c-addr* for *len* bytes. Return the beginning of the destination string (*c-addr*) and the length of the resulting string (*len* plus 1). There must be room at the end of the destination string to hold the additional character.
F9 7A MINUSTRAILING
(c-addr len$_1$ — c-addr len$_2$)
If *len$_1$* is greater than zero, *len$_2$* is equal to *len$_1$* less the number of spaces (ASCII 20$_h$) at the end of the character string specified by *c-addr len$_1$*. If *len$_1$* is zero or the entire string consists of spaces, *len$_2$* is zero.
F9 7B MINUSZEROS
(c-addr len$_1$ — c-addr len$_2$)
If *len1* is greater than zero, *len$_2$* is equal to *len$_1$* less the number of nulls (ASCII 0h) at the end of the character string specified by *c-addr len$_1$*. If *len$_1$* is zero or the entire string consists of nulls, *len$_2$* is zero.
F9 7C STORECOUNT
(char c-addr —)
Store number *char* to the byte at *c-addr*. Generate a STRING_TOO_LARGE throw code if *char* is larger than 255.

1.8.2 TLV Buffer Access

F9 80 TLV
(num — c-addr len fmt)
Return the access parameters for the TLV whose Tag is *num*. This may generate an UNDEFINED_TLV throw code.

APPENDIX

F9 81 TLVFETCH

(c-addr$_1$ len$_1$ fmt — num | c-addr$_2$ len$_2$)

Return the contents of the internal TLV buffer according to its TYPE field which is the lower eight bits of *fmt*. Type codes 0 and 2 return numbers on the stack, while the others return string pointers. The address returned by type code 3 fields is temporary and must be moved immediately to a more permanent location. The *len$_2$* returned for strings is the same as that last stored in the buffer.

F9 82 TLVSTORE

(num c-addr$_2$ len$_2$ fmt | c-addr$_1$ len$_1$ c-addr$_2$ len$_2$ fmt —)

Set the contents of the internal TLV buffer according to its TYPE field which is the lower eight bits of *fmt*. Type codes 0 and 2 take numbers on the stack, while the others take string pointers. This action will set the parsed status bit for this TLV.

F9 83 TLVBITFETCH

(c-addr — flag)

Return the results of masking the contents of the internal TLV buffer referenced by the sequence at *c-addr* against the Value field at that location. This may generate an UNDEFINED_TLV throw code. The *flag* will be returned TRUE if all the bits defined in the mask are on in the internal buffer. Otherwise, FALSE is returned. Only the bytes covered by the shorter of the two locations are checked.

F9 84 TLVBITSTORE

(flag c-addr —)

Set the contents of the internal TLV buffer referenced by the sequence at *c-addr* based upon the Value field at that location. If *flag* is FALSE (0), then all the bits defined there will be turned off. Otherwise, they will all be turned on.

1.8.3 TLV Processing

F9 85 PARSETLV

(c-addr len —)

Process *len* bytes at *c-addr* for TLV sequences. This may generate an UNDEFINED_TLV throw code. Each Tag field encountered will place Length field bytes from its Value field into its internal buffer and set its parsed status bit. When a constructed Tag field is encountered, all the internal TLV buffers that have been defined as being associated with it are cleared before the Value field is parsed for TLV sequences. No exception will be generated if a TLV is encountered in a constructed template to which it has not been defined as being associated with.

F9 86 PLUSDOL

(c-addr$_1$ len$_1$ c-addr$_2$ len$_2$ — c-addr$_2$ len$_3$)

Process *len$_1$* bytes at *c-addr$_1$* for Tag and Length fields. This may generate an UNDEFINED_TLV throw code. Each Tag field encountered will place Length field bytes from its internal buffer into a Value field at the end of the output string at *c-addr$_2$* for *len$_2$* bytes. Return the beginning of the destination string (*c-addr$_2$*) and the sum of the two lengths (*len$_3$*). There must be room at the end of the output string to hold both strings.

F9 87 PLUSTLV

(c-addr len$_1$ num — c-addr len$_2$)

Add the TLV sequence whose Tag is *num* to the end of the output string at *c-addr* for *len$_1$* bytes. This may generate an UNDEFINED_TLV throw code. The Tag, Length and Value fields are formatted according to TLV rules based upon the data in its internal buffer. Return the beginning of the destination string (*c-addr*) and the sum of the two lengths (*len$_2$*). There must be room at the end of the output string to hold both strings.

F9 89 TLVSTATUS

(fmt — num char)

Decode the status of the TLV access parameter *fmt*. The returned *num* is the format indicator 0- and bits in the returned *char* have the following significance, where bit 0 is the least significant bit:

0	0 = unparsed, 1 = parsed
1-7	Reserved for future use

1.8.4 TLV Sequence Access

F9 8A STOREBCD
(u c-addr len —)
Store number *u* as a Binary Coded Decimal sequence into the string at *c-addr* for *len* bytes. The number is formatted with each digit representing 4-bit nibbles in the output string. Leading nibbles will be filled with 0's if needed. The most significant part of the number will be truncated if *len* is not long enough to hold all the digits.
F9 8B FETCHBCD
(c-addr len — u)
Fetch number *u* from a Binary Coded Decimal sequence at *c-addr* for *len* bytes. The number is formatted with each digit representing 4-bit nibbles in the input string. A DIGIT_TOO_LARGE exception is thrown if any nibble is not a valid BCD digit.
F9 8C STOREBN
(u c-addr len —)
Store number *u* as a Binary Number into the string at *c-addr* for *len* bytes. The Most Significant Byte of the number is stored first. Leading bytes will be filled with 0's if needed. The most significant part of the number will be truncated if *len* is not long enough to hold all the bytes.
F9 8D FETCHBN
(c-addr len — u)
Fetch number *u* as a Binary Number from the string at *c-addr* for *len* bytes. The most significant byte of the number is fetched first. If there are more than four bytes of data at that location, the most significant bytes will be lost.
F9 8E STORECN
($c\text{-}addr_1$ len_1 $c\text{-}addr_2$ len_2 —)
Store the number at $c\text{-}addr_1$ for len_1 bytes as a Compressed Number into the string at $c\text{-}addr_2$ for len_2 bytes. The number is formatted with each character representing a 4-bit nibble in the output string. Trailing nibbles will be filled with F's if needed. The number will be truncated if len_2 is not long enough to hold all the characters ($len_2 < [len_1 + 1]/2$). A DIGIT_TOO_LARGE throw code will be generated if a character in the input string is not a number.
F9 8F FETCHCN
($c\text{-}addr_1$ len_1 — $c\text{-}addr_2$ len_2)
Fetch a string to the temporary location $c\text{-}addr_2$ for len_2 bytes that represents the Compressed Number in the string at $c\text{-}addr_1$ for len_1 bytes. The number is formatted with each character of the output string representing a 4-bit nibble in the input string. The output string will be terminated when a nibble with all bits set or the end of the string is encountered. A DIGIT_TOO_LARGE throw code will be generated if a nibble in the input string is not a number. The output string must be moved to a more permanent location immediately.
F9 90 TLVFETCHNAME
($c\text{-}addr_1$ — $c\text{-}addr_2$ num)
Parse the TLV sequence at $c\text{-}addr_1$ for a Tag field. Return the address $c\text{-}addr_2$, which is past the Tag field, and the *num* of the Tag field.
F9 91 TLVFETCHLENGTH
($c\text{-}addr_1$ — $c\text{-}addr_2$ len)
Parse the TLV sequence at $c\text{-}addr_1$ for a Length field. Return the address $c\text{-}addr_2$, which is past the Length field, and the *len* contained in that field.

1.9 Module Handling

The following tokens provide for the storage and execution of EPICode modules in the Virtual Machine.

F9 A0 EXECUTEMODULE
(c-addr len — flag)
Load a module from the module directory using the AID specified by *c-addr len*. Exception CANNOT_LOAD_MODULE is thrown if an error occurs. *flag* is TRUE if the module is not found, FALSE

being "loaded successfully".

F9 A1 INITMODULEBUFFER

(—)

Prepare for acquisition of a new module.

F9 A2 MODULEBUFFERAPPEND

(c-addr len —)

Append the contents of the buffer defined by *c-addr* and *len* to the module acquisition buffer. Exception CANNOT_ADD_TO_MODULE is thrown if the module buffer has not been prepared, or if the module buffer capacity is exceeded.

F9 A3 REGISTERMODULE

(c-addr len —)

Register the module buffer in the module directory under the given EPICode AID specified by *c-addr len*. The resources associated with managing the module buffer are automatically released.

F9 A4 RELEASEMODULEBUFFER

(—)

Release the resources used by the internal module buffer. This is required if premature loading of a module must be terminated by the application without registering the module in the module directory.

F9 A5 DELETEMODULE

(c-addr len — flag)

Delete the module whose AID is specified by *c-addr len* from the module directory. *flag* is zero if the operation succeeded.

F9 A6 MODULEINFO

($c\text{-}addr_1$ len_1 — $c\text{-}addr_2$ len_2 flag)

Return "public" information on the module which is registered in the module directory under the AID specified by $c\text{-}addr_1$ len_1. *flag* is zero if the operation succeeded and the data at $c\text{-}addr_1$ is valid. The structure of the buffer returned by this token is defined by the module header information. Only the entries EPF_VER through EPF_ENTRY are returned by this function.

F9 A7 LOADCARDMODULE

(a-addr—)

Load the module at *a-addr*. *a-addr* is the address of the header of an EPICode module delivered from the card into internal storage. The exception BAD_CARD_MODULE is thrown if the module violates any precondition for card module loading.

F9 A8 MODULESCHANGED

(— u)

Return a value *u* indicating whether modules have changed. Bits 0 through 7 define which module classes have been registered in the module directory since the last execution of this token. For example, a module registered with an initial AID byte which is F4 will set bit 4 in the return status. Bits 8 through 31 are reserved for future expansion.

1.10 Extensible Memory Handling

The following tokens provide access to an extensible "rubber band" buffer of linear memory in the data space provided and managed by the virtual machine.

F9 B0 EXTEND

(len — a-addr)

Extend the "rubber band" buffer by *len* cells, returning the cell-aligned address *a-addr* of the first cell in the allocated buffer. ZERO EXTEND returns a pointer to the next unallocated cell. An OUT_OF_MEMORY exception is thrown if there is insufficient memory available.

F9 B1 BEXTEND

(len — c-addr)

Extend the "rubber band" buffer by *len* bytes, returning the address c-*addr* of the first byte in the allocated buffer. ZERO BEXTEND returns a pointer to the next unallocated byte. An OUT_OF_MEMORY exception is thrown if there is insufficient memory available.

F9 B2 RELEASE
(addr —)
Release storage acquired through EXTEND or BEXTEND, setting the "free pointer" to *addr*. If *addr* is invalid (before the start of the rubber-band buffer, or beyond the current "free pointer") an ANS exception -9 (invalid memory address) is thrown.

Further tokens:

F9 B0 DSCHECK
(u — flag)
Check that there are at least *u* data cells remaining on the data stack. Return FALSE if this is the case, otherwise TRUE.

F9 B1 RSCHECK
(u — flag)
Check that there are at least *u* data cells remaining on the return stack. Return FALSE if this is the case, otherwise TRUE.

1.11 Security Commands

Security algorithm processing may occupy several seconds on some terminals. The present invention includes that the present single SECALGO command be factored into initiation and completion components to facilitate use with multitasking implementations. This is under investigation, and the following proposals are made as an alternative to SECALGO.

F9 56 SECALGOBEGIN
(c-addr$_1$ len c-addr$_2$ num — flag)
This computes using the algorithm of type *num*. *c-addr$_1$* is the input data buffer for computation, and *len* its length. *c-addr$_2$* is the output buffer for storage of the result. This function returns a *flag* indicating FALSE is computation could be initiated successfully.

F9 57 SECALGOEND
(— ior)
This function returns an *ior* indicating: 0 = computation completed successfully; -1 = computation still in progress; 1 = computation failed

Exception Codes

This section includes all codes used as arguments to the standard exception handling function THROW.

The following table shows the ANS Forth codes used in EPIC kernels.

#	Reserved for	#	Reserved for
-3	stack overflow	-23	address alignment exception
-4	stack underflow	-24	invalid numeric argument
-5	return stack overflow	-25	return stack imbalance
-6	return stack underflow	-26	loop parameters unavailable
-7	do loops nested too deeply during execution	-27	invalid recursion
-9	invalid memory address	-28	user interrupt
-10	division by zero	-36	invalid file position
-11	result out of range	-37	file I/O exception
-12	argument type mismatch	-38	non existent file
-17	pictured numeric output string overflow	-39	unexpected end of file
-20	write to a read only location	-53	exception stack overflow
-21	unsupported operation	-57	exception sending or receiving a character

Claims

1. A transaction management system for executing transactions between a first device and a second device, said first and second devices being adapted for communication with each other and at least one of said first and second devices being an integrated circuit card, said system comprising: at least one input/output device;
a portable virtual machine for interpreting a computer program on said first device, said virtual machine comprising a virtual microprocessor and a driver for said at least one input/output device; and
execution means responsive to said interpreted program for executing said program.

2. A terminal comprising a first device for carrying out a transaction with a second device, at least one of said first and second devices being an integrated circuit card, comprising:
a portable virtual machine interpreting a computer program on said first device, said portable virtual machine comprising a virtual microprocessor and a driver for at least one input/output device, and
execution means responsive to said interpreted program for executing said program.

3. A self-contained portable intelligent card including a first device for carrying out a transaction with a second device, said intelligent card comprising:
a portable virtual machine comprising a virtual microprocessor and a driver for at least one input/output device.

4. A system according to claim 1 or a terminal according to claim 2 or an intelligent card according to claim 3, wherein the machine instructions of said virtual machine are a set of tokens, said tokens being customized byte code.

5. An intelligent card according to claim 3 or 4, further comprising a computer program stored on said intelligent card, said virtual machine interpreting said computer program and execution means responsive to said interpreted program for executing said program.

6. System or terminal according to claim 4 or intelligent card according to 5, wherein said computer program is written as a stream of tokens selected from said set of tokens and corresponding in-line data.

7. System or terminal or intelligent card according to claim 6, wherein said stream of tokens is transported in a module, a module including the stream of tokens together with the corresponding in-line data required for execution of the module.

8. System or terminal or intelligent card according to claim 7 wherein said module also includes an indication of the memory requirements for execution of said module.

9. System or terminal or intelligent card according to claim 8, wherein the virtual machine also includes a means for loading said module and interpreting the tokens therein.

10. System or terminal or intelligent card according to claim 9, wherein said means for token loading and interpreting reads said tokens in said module and an exception is thrown if a token is read which does not belong to said set.

11. System or terminal or intelligent card according to any of claims 7 to 10, wherein said virtual machine includes read/writable logical address space having a repository for at least said module, said module includes an indication of the amount of read/writable logical address space needed for its execution; and
said virtual machine also includes means for allocating, on loading of said module, an amount of read/writable logical address space in accordance with said indication, said allocated read/writable logical address space has defined and protected boundaries.

12. System or terminal or intelligent card according to claim 11, further comprising means for deallocating said allocated amount of read/writable logical address space on termination of said module.

13. System or terminal or intelligent card according to any of claims 9 to 12, wherein said second device includes means for providing at least one program instruction capable of at least modifying the execution time behavior of said computer program; and, after said means for loading and interpreting has loaded said module and while said module is running, said means for loading and interpreting loads and interprets said at least one program instruction dependent upon a pre-defined security condition; and
said execution means responds to said loaded program instruction as interpreted by said virtual machine and executes said computer program with said modified behavior.

14. System or terminal or intelligent card according to claim 13, wherein said security condition is provided by a function.

15. System or terminal or intelligent card according to any of claims 9 to 14, further comprising read/writable logical address space including at least one database including at least one record, said module includes an indication of the amount of uninitialised read/writable logical address space necessary for execution of said module;
said loader allocating the required amount of uninitialised logical address space in accordance with said indication; and
means for accessing a record in said database, said record in said database only being accessible through said module, and said accessing means providing a window onto a current record of said database and copying said record into a portion of said uninitialised read/writable logical address space addressable by said application program.

16. A transaction management system comprising:
a first device and a second device, said first and second devices being adapted for communication with each other, at least one of said first and second devices being an integrated circuit card;
said second device including means for providing at least one program instruction capable of at least modifying the execution time behavior of a computer program on said first device;
said first device including a virtual machine, said virtual machine comprising means for loading and interpreting said computer program, said means for loading and interpreting being further adapted to load and interpret said at least one program instruction dependent upon a pre-defined security condition after said means for loading and interpreting has loaded said computer program and while said computer program is running; and
execution means for executing said loaded and interpreted computer program with said modified behavior in response to said loaded and interpreted program instruction.

17. A terminal comprising a first device for carrying out a transaction with a second device and at least one of said first and second devices being an integrated circuit card, said second device including means for providing at least one program instruction capable of at least modifying the execution time behavior of a computer program on said first device;

said terminal comprising:

said first device including a virtual machine, said virtual machine comprising means for loading and interpreting said computer program, said means for loading and interpreting being further adapted to load and interpret said at least one program instruction dependent upon a pre-defined security condition after said means for loading and interpreting has loaded said computer program and while said computer program is running; and

execution means for executing said loaded and interpreted computer program with said modified behavior in response to said loaded and interpreted program instruction.

18. A self-contained portable intelligent card including a first device for carrying out a transaction with a second device, said second device including means for providing at least one program instruction capable of at least modifying the execution time behavior of a computer program on said first device, said intelligent card comprising:

said first device including a virtual machine, said virtual machine comprising means for loading and interpreting said computer program, said means for loading and interpreting being further adapted to load and interpret said at least one program instruction dependent upon a pre-defined security condition after said means for loading and interpreting has loaded said computer program and while said computer program is running; and

execution means for executing said loaded and interpreted computer program with said modified behavior in response to said loaded and interpreted program instruction.

19. A system according to claim 16 or a terminal according to claim 17 or an intelligent card according to claim 18, wherein said security condition is provided by a function.

20. System or terminal or intelligent card according to claim 19, wherein said at least one program instruction is a first program instruction and said first device includes a second program instruction capable of at least modifying the execution time behavior of said computer program, said first program instruction including a reference to said second program instruction; and

said means for loading and interpreting are responsive to said reference to load said second program instruction, said execution means executing said computer program with said modified behavior as determined by said second program instruction.

21. System or terminal or intelligent card according to claim 20, wherein said computer

program and said first and second program instructions are written in terms of a stream of tokens and corresponding in-line data, each token being a customized byte code selected from a set of customized byte codes.

22. System or terminal or intelligent card according to claim 21, said virtual machine vectors said stream of tokens and said in-line data of said first and second program instructions into said stream of tokens of said computer program.

23. System or terminal or intelligent card according to claim 21 or 22, wherein at least the streams of tokens of said computer program and at least said second program instruction are each transported in a module, each module including the relevant stream of tokens together with the corresponding in-line data required for execution of said module.

24. System or terminal or intelligent card according to claim 23, wherein said module also includes an indication of the memory required for execution of said module.

25. System or terminal or intelligent card according to claim 23 or 24, wherein the module of said computer program also includes an exclusive list of at least one modifiable socket, said at least one socket defining the position in the stream of tokens and in-line data of said computer program module into which said virtual machine vectors said first program instruction.

26. System or terminal or intelligent card according to claim 25, wherein said at least one modifiable socket in said computer program module contains an execution vector to a default behavior.

27. System or terminal or intelligent card according to claim 26, wherein said execution means executes said computer program with said default behavior if said pre-defined security condition does not allow loading of said at least one program instruction.

28. System or terminal or intelligent card according to any of claims 23 to 27, wherein said virtual machine includes read/writable logical address space, said module includes an indication of the amount of read/writable logical address space needed for its execution and;

said virtual machine also includes means for allocating, on loading of said computer program module, an amount of read/writable logical address space in accordance with said indication, said allocated read/writable logical address space having defined and protected boundaries; and

means for deallocating said amount of read/writable logical address space on termination of said computer program module.

29. System or terminal or intelligent card according to any of claims 23 to 28, further comprising read/writable logical address space including at least one database including a plurality of records, said module includes an indication of the amount of uninitialised read/writable logical address space necessary for execution of said module;

said means for loading allocating the required amount of uninitialised logical address space in accordance with said indication; and

means for accessing a record in said database, records in said database only being accessible through said module, said accessing means providing a window onto a current record of said database and for copying said record into a portion of said uninitialised read/writable logical address space addressable by said application program.

30. System or terminal or intelligent card according to any of claims 16 to 27, wherein said security condition includes means for verifying at least the origin and integrity of data and program instructions on said second device.

31. A transaction system for executing transactions between a first device and a second device, said system comprising: a virtual machine for interpreting a set of customized byte code tokens applied thereto;

said virtual machine including a virtual processing unit and read/writable logical address space;

at least one first application program including an indication of the amount of read/writable logical address space needed for its execution, said at least one first application program being written as a stream of tokens selected from said set of tokens and corresponding in-line data;

said virtual machine also including:

a loader for loading said at least one first application program; and

means for allocating a first amount of read/writable logical address space specifically for

said at least one first application program in accordance with said indication, said allocated read/writable logical address space having defined and protected boundaries.

32. A terminal comprising a first device for executing transactions with a second device, said first device comprising: a virtual machine for interpreting a set of customized byte code tokens applied thereto;
said virtual machine including a virtual processing unit and read/writable logical address space;
at least one first application program including an indication of the amount of read/writable logical address space needed for its execution, said at least one first application program being written as a stream of tokens selected from said set of tokens and corresponding in-line data;
said virtual machine also including:
a loader for loading said at least one first application program; and
means for allocating a first amount of read/writable logical address space specifically for said at least one first application program in accordance with said indication, said allocated read/writable logical address space having defined and protected boundaries.

33. A self-contained portable intelligent card including a first device for carrying out a transaction with a second device, said first device comprising: a virtual machine for interpreting a set of customized byte code tokens applied thereto;
said virtual machine including a virtual processing unit and read/writable logical address space;
at least one first application program including an indication of the amount of read/writable logical address space needed for its execution, said at least one first application program being written as a stream of tokens selected from said set of tokens and corresponding in-line data;
said virtual machine also including:
a loader for loading said at least one first application program; and
means for allocating a first amount of read/writable logical address space specifically for said at least one first application program in accordance with said indication, said allocated read/writable logical address space having defined and protected boundaries.

34. System according to claim 31 or terminal according to claim 32 or intelligent card

according to claim 33, further comprising means for explicitly deallocating said first amount of read/writable logical address space on termination of said at least one program.

35. System or terminal or intelligent card according to any of claims 31 to 34, wherein at least one of the first and second devices is an ICC.

36. System or terminal or intelligent card according to any of the claims 31 to 35, wherein said first application program also includes a first exclusive list of at least one function which can be exported to other application programs, further comprising means for making said at least one function available to other programs.

37. System or terminal or intelligent card according to any of claims 31 to 36, wherein said first application program is a first module and said other application programs are other modules, each module including at least a stream of tokens selected from said set of tokens, corresponding in-line data, a first exclusive list of at least one function to be exported and an indication of the amount of read/writable logical address space needed for execution of the module.

38. System or terminal according to claim 37, wherein said first module includes a second exclusive list identifying at least one second module from which at least one function is to be imported, and said loader loading said at least second module in accordance with said second list on loading said first module.

39. System or terminal or intelligent card according to claim 38, wherein said first module is terminated if said at least one second module to be imported does not load successfully.

40. System or terminal or intelligent card according to any of claims 37 to 39, wherein said means for allocating allocates said first amount of read/writable logical address space on loading said first module and only allocates a second or further amount of read/writable logical address space for said first module in a single extensible buffer, starting at a first address; and

on release of said second amount of read/writable logical address space by said first module, said deallocating means deallocates said second amount of read/writable logical address space and all further allocations beyond said first address.

41. System or terminal or intelligent card according to claim 40, wherein said deallocating means deallocates said second amount of read/writable logical address and all further allocations beyond said first address on termination of said first module.

42. System or card or intelligent card according to any of claims 37 to 41, wherein said read/writable logical address space includes at least one database including at least one record, said module includes an indication of the amount of uninitialised read/writable logical address space necessary for execution of said module, and records in said database only being accessible through said module;
said loader allocating the required amount of uninitialised logical address/space in accordance with said indication; and
means for accessing a record in said database, said accessing means providing a window onto a current record of said database and copying said record into a portion of said uninitialised read/writable logical address space addressable by said application program.

43. System or terminal or intelligent card according to any of claims 34 to 42 wherein said deallocating means erases any amount of allocated read/writable logical address space on termination of said first module.

44. A transaction system for executing transactions between a first device and a second device, at least one of said first and second devices being an integrated circuit card, said system comprising: a virtual machine for interpreting a set of customized byte code tokens applied thereto;
said virtual machine including a virtual processing unit and read/writable logical address space;
at least one database including at least one record and at least one computer program for execution by said virtual machine, said computer program being a module written in terms of a stream of said tokens selected from said set and including an indication of the amount of uninitialised read/writable logical address space necessary for execution of said module;
a loader for loading said module and for allocating the required amount of uninitialised logical address/space in accordance with said indication; and
means for accessing a record in said database, records in said database only being accessible through said module, and said accessing means providing a window onto a

current record of said database and copying said record into a portion of said uninitialised read/writable logical address space addressable by said application program.

45. A terminal comprising a first device for executing transactions with a second device, at least one of said first and second devices being an integrated circuit card, said first device comprising: a virtual machine for interpreting a set of customized byte code tokens applied thereto;

said virtual machine including a virtual processing unit and read/writable logical address space;

at least one database including at least one record and at least one computer program for execution by said virtual machine, said computer program being a module written in terms of a stream of tokens selected form said set and including an indication of the amount of uninitialised read/writable logical address space necessary for execution of said module;

a loader for loading said module and for allocating the required amount of uninitialised logical address/space in accordance with said indication; and

means for accessing a record in said database, records in said database only being accessible through said module, and said accessing means providing a window onto a current record of said database and copying said record into a portion of said uninitialised read/writable logical address space addressable by said application program.

46. A self-contained portable intelligent card including a first device for carrying out a transaction with a second device,

said first device comprising:

a virtual machine for interpreting a set of customized byte code tokens applied thereto;

said virtual machine including a virtual processing unit and read/writable logical address space;

at least one database including at least one record and at least one computer program for execution by said virtual machine, said computer program being a module written in terms of a stream of tokens selected form said set and including an indication of the amount of uninitialised read/writable logical address space necessary for execution of said module;

a loader for loading said module and for allocating the required amount of uninitialised logical address/space in accordance with said indication; and

means for accessing a record in said database, records in said database only being accessible through said module, said accessing means providing a window onto a current

record of said database and for copying said record into a portion of said uninitialised read/writable logical address space addressable by said application program.

47. A system according to claim 44 or a terminal according to claim 45 or an intelligent card according to claim 46, wherein said database is instantiated on first loading of said module.

48. System or terminal or intelligent card according to any of the previous claims 1 to 47, wherein said virtual machine is a stack machine.

49. System or terminal or intelligent card according to claim 48, wherein said virtual machine is at least a two stack machine, in which the first stack is a data stack and the second stack is a return stack.

50. System or terminal or intelligent card according to any of the preceding claims 1 to 49, wherein said virtual machine includes a local variable frame memory, a frame pointer register for storing a frame pointer pointing to the frame start in memory and a frame end pointer register pointing to the frame end in memory.

51. System or terminal or intelligent card according to claim 49 or 50, wherein said data and return stacks are not in a memory directly addressable by said computer program but are only accessible via stack operations defined by tokens and interpreted by said virtual machine.

52. System or terminal according to any previous claim wherein said first device is a hand-held device.

53. System or terminal according to claim 52, wherein said hand-held device includes an Integrated Circuit Card (ICC).

54. System or terminal according to any of the preceding claims 1 to 53, wherein said second device comprises an ICC.

55. System according to any previous claim 1 to 54, wherein said first device is a terminal.

56. System or terminal or intelligent card according to any of the previous claims 1 to 55 wherein, both said first and second devices include ICC's.

57. System or terminal or intelligent card according to any previous claim 1 to 56, wherein the transaction comprises at least one execution of the following sequence:

a. creating a communications link between said first and second device;

b. selection of an application including said computer program and the associated data set that defines the transaction;

c. execution of said application; and

d. termination of the transaction.

58. System or terminal or intelligent card according to any previous claim 1 to 57, wherein the transaction is a financial transaction and said system is a financial transaction management system.

59. An integrated circuit card containing a program instruction capable of modifying the behavior of a program running in the system according to any of claims 1 to 58, in a terminal according to any of the claims according to any of claim 2 to 58 or an intelligent card according to any of claims 3 to 58.

60. A method of carrying out a transaction between a first device and a second device, at least one of said first and second devices being an integrated circuit card; comprising providing at least one program instruction on said second device capable of at least modifying the execution time behavior of a computer program on said first device;
loading and interpreting said computer program,
loading and interpreting said at least one program instruction dependent upon a pre-defined security condition while said computer program is running; and
executing said loaded and interpreted computer program with said modified behavior in response to said loaded and interpreted program instruction.

61. A method of carrying out a transaction between a first device and a second device, comprising:
interpreting at least one application program written as a stream of byte code tokens

selected from a set of tokens and corresponding in-line data;

loading said at least one application program;

allocating a first amount of read/writable logical address space specifically for said at least one application program in accordance with an indication contained within said application program of the amount of read/writable logical address space needed for its execution; and

defining and protecting the boundaries of said allocated read/writable logical address space.

62. A method according to claim 61, further comprising:

explicitly deallocating said first amount of read/writable logical address space on termination of said at least one first application program.

63. A method of carrying out a transaction system between a first device and a second device, at least one of said first and second devices being an integrated circuit card, comprising: interpreting the tokens in a module written in terms of a stream of said tokens selected from a set of tokens;

allocating an amount of uninitialised logical address/space in accordance with an indication in said module of the amount of uninitialised read/writable logical address space necessary for execution of said module;

accessing a record in a database by providing a window onto a current record of said database, records in a database only being accessible through said module; and

copying said record into a portion of said uninitialised read/writable logical address space addressable by said module.

64. A method of carrying out a transaction between a first device and a second device, at least one of said first and second devices being an integrated circuit card, comprising:

providing a portable virtual machine comprising a virtual microprocessor and a driver for at least one input/output device;

interpreting a computer program on said first device using said portable virtual machine; and

executing said program in response to said interpreted program.

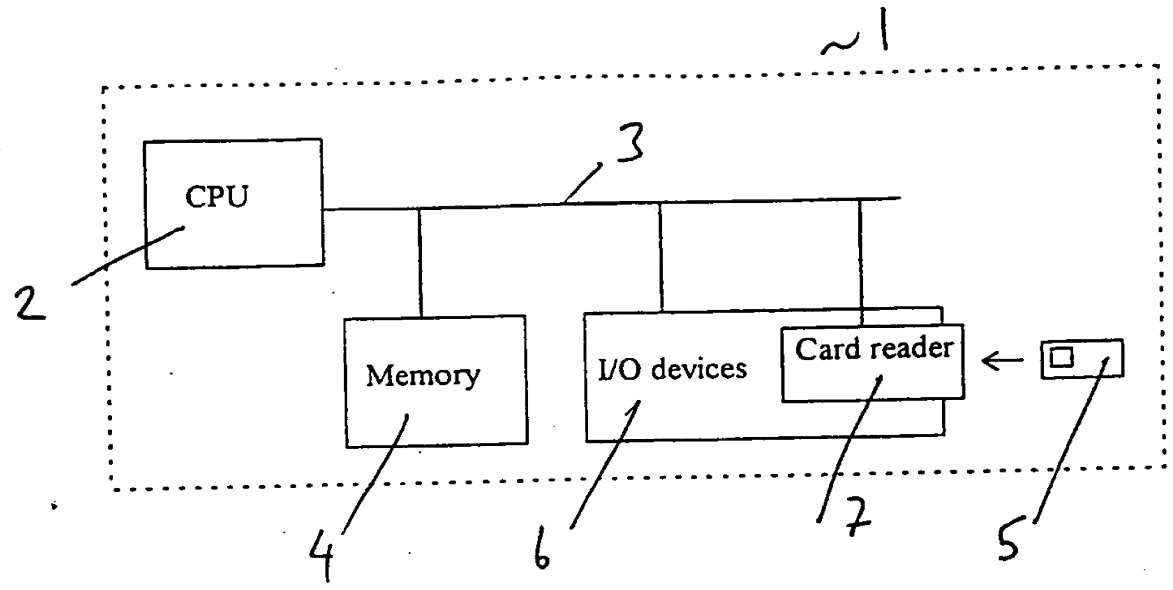

1
CPU
2
3
Memory
I/O devices
Card reader
4
6
7
5

Fig. 1

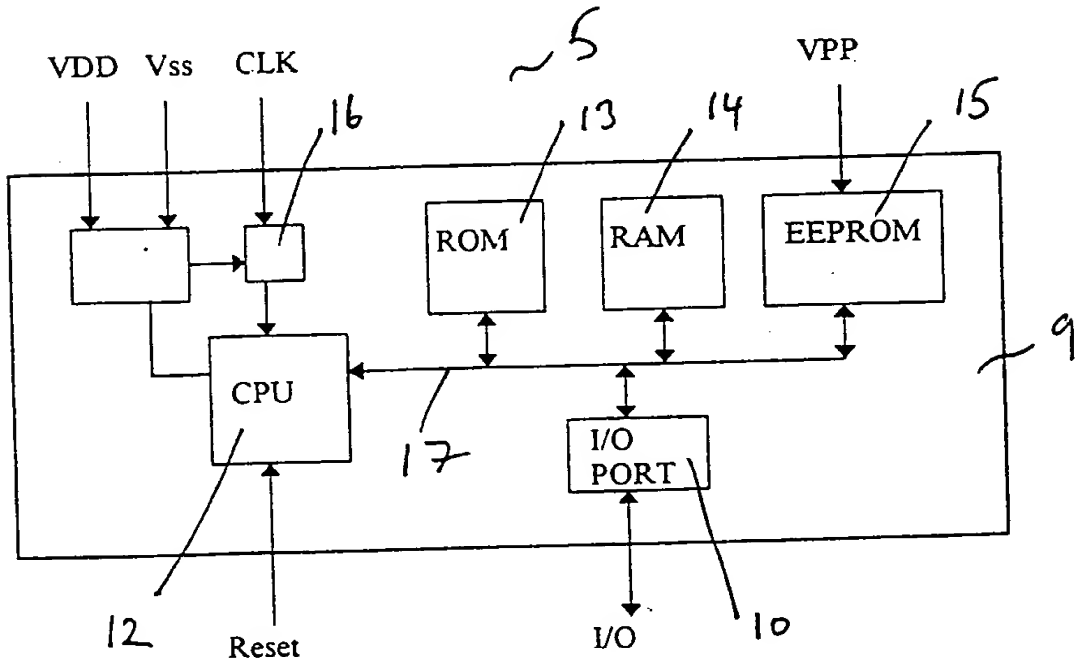

VDD
Vss
CLK
5
VPP
16
13
14
15
ROM
RAM
EEPROM
CPU
9
17
I/O
PORT
12
Reset
I/O
10

Fig. 2

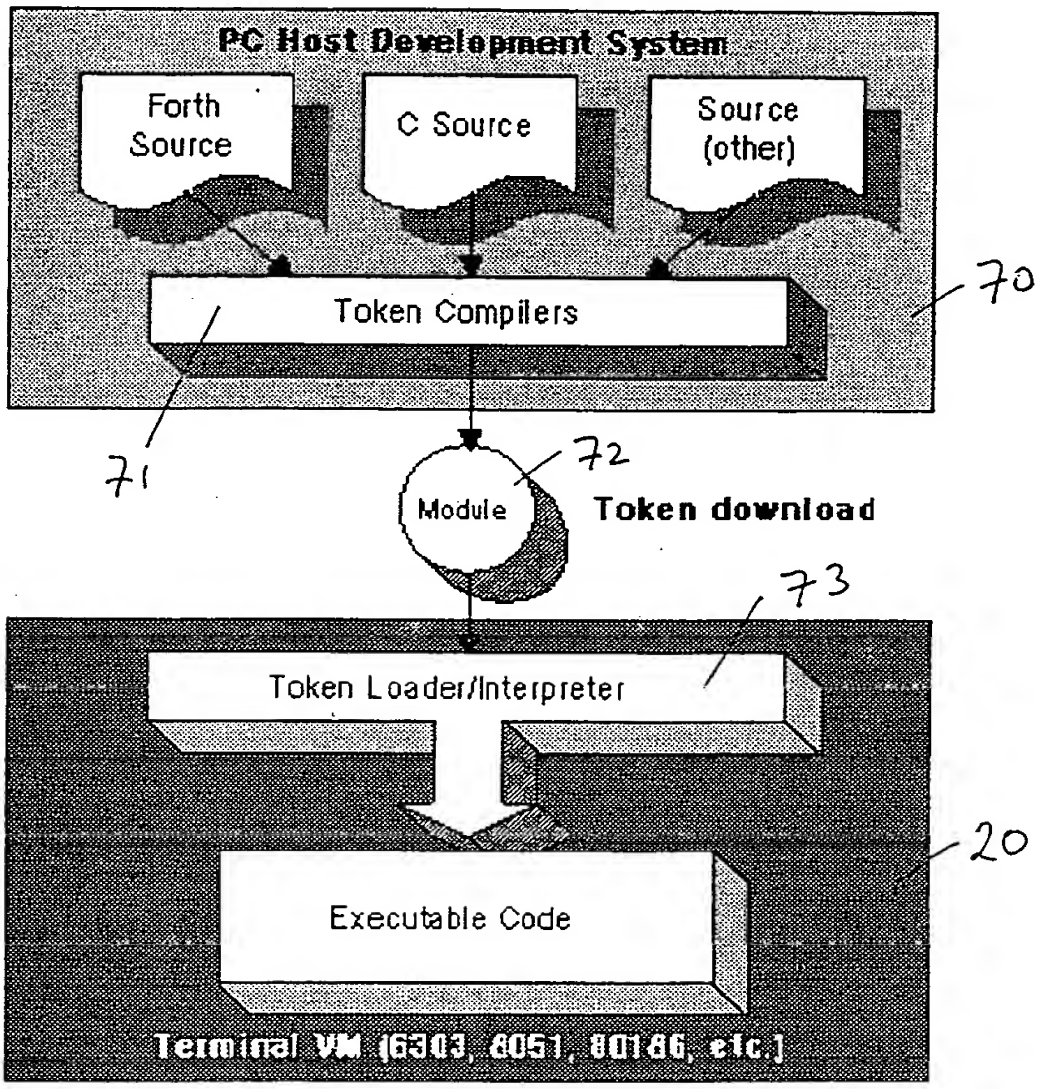

PC Host Development System
Forth Source
C Source
Source (other)
Token Compilers
70
71
72
Module
Token download
73
Token Loader/Interpreter
20
Executable Code
Terminal VM (6303, 8051, 80186, etc.)

Fig. 3

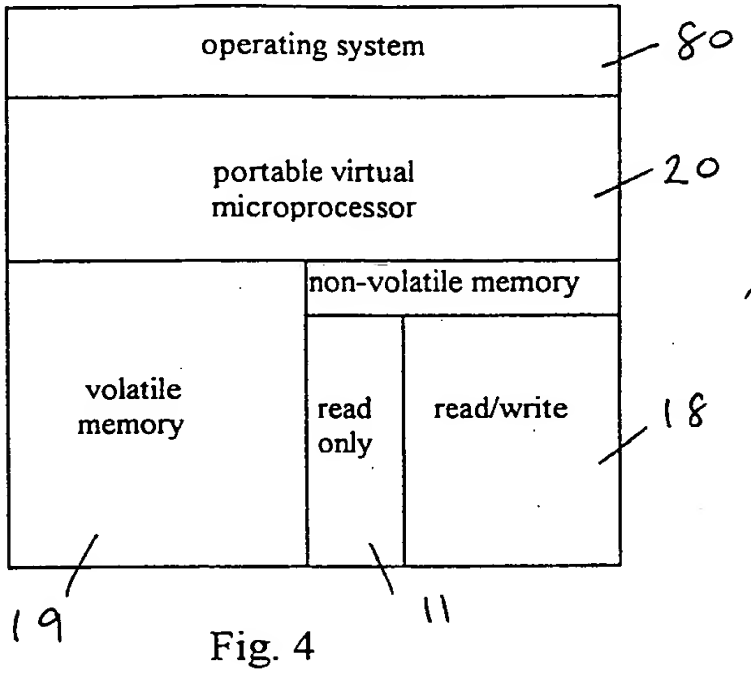

operating system
80
portable virtual microprocessor
20
non-volatile memory
~1
volatile memory
read only
read/write
18
19
11

Fig. 4

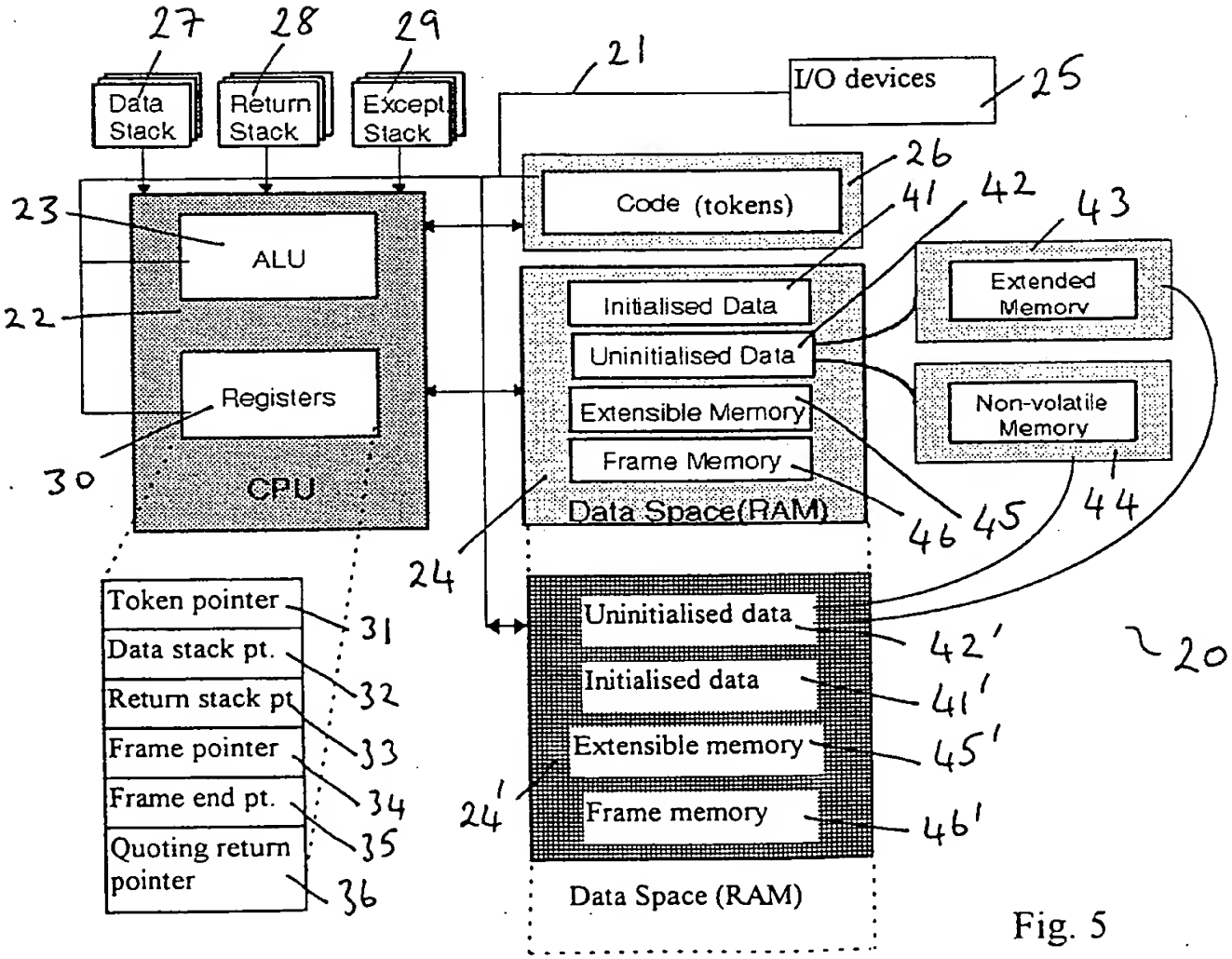

27
28
29
21
I/O devices
25
Data Stack
Return Stack
Except Stack
26
Code (tokens)
41
42
43
23
ALU
Initialised Data
Extended Memory
22
Uninitialised Data
Registers
Extensible Memory
Non-volatile Memory
30
CPU
Frame Memory
Data Space(RAM)
46
45
44
24
Token pointer
31
Uninitialised data
42'
~20
Data stack pt.
32
Initialised data
41'
Return stack pt
33
Extensible memory
45'
Frame pointer
34
24'
Frame memory
46'
Frame end pt.
35
Quoting return pointer
36
Data Space (RAM)

Fig. 5

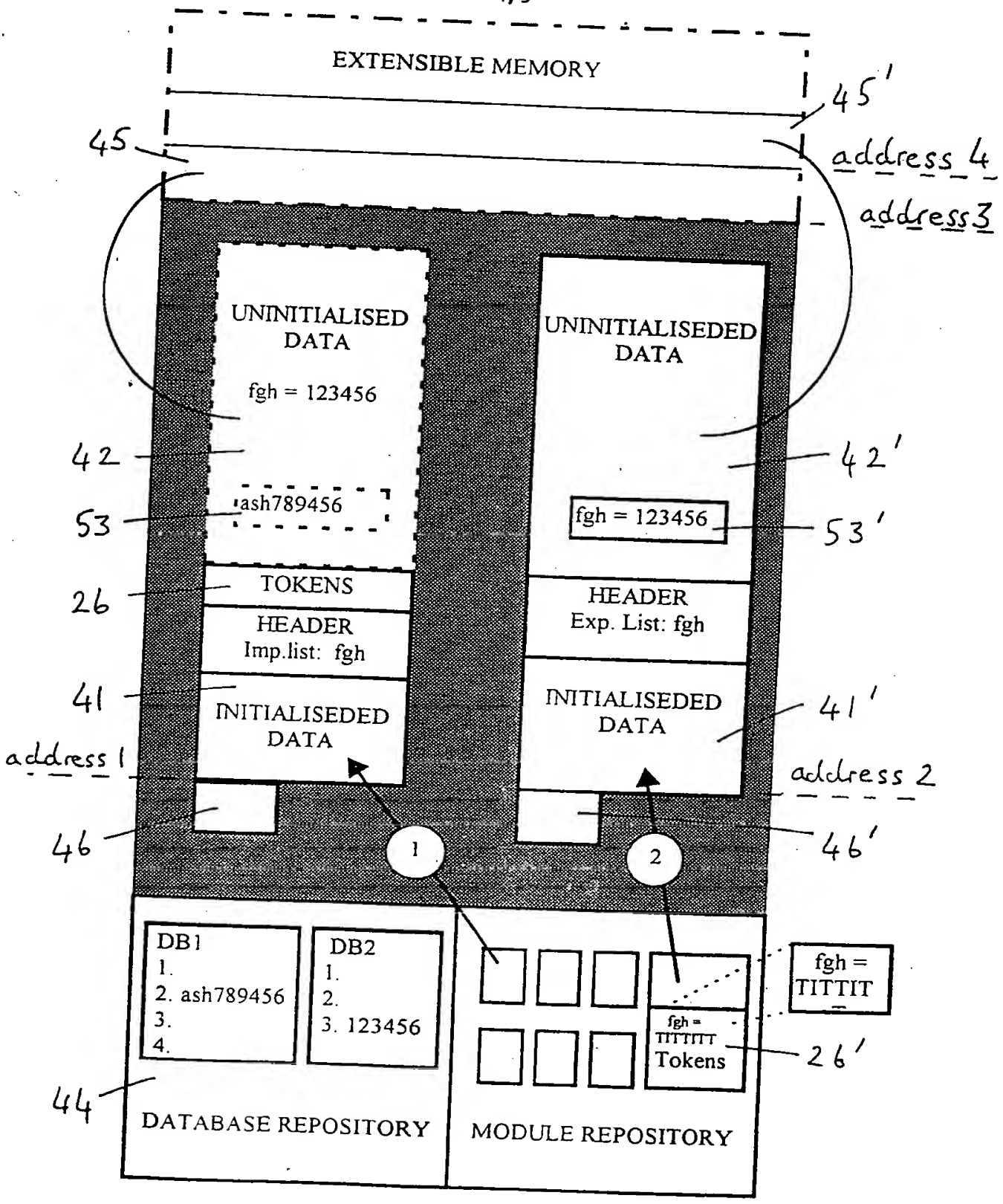

EXTENSIBLE MEMORY
45'
45
address 4
address 3
UNINITIALISED DATA
fgh = 123456
UNINITIALISEDED DATA
42
42'
ash789456
53
fgh = 123456
53'
TOKENS
26
HEADER
Imp.list: fgh
HEADER
Exp. List: fgh
41
INITIALISEDED DATA
INITIALISEDED DATA
41'
address 1
address 2
46
46'
1
2
DB1
1.
2. ash789456
3.
4.
DB2
1.
2.
3. 123456
fgh = TITTIT
fgh = TITTITT
Tokens
26'
44
DATABASE REPOSITORY
MODULE REPOSITORY
direct access by module 1 denied
direct access by module 1 allowed
direct access by modules allowed

Fig. 6

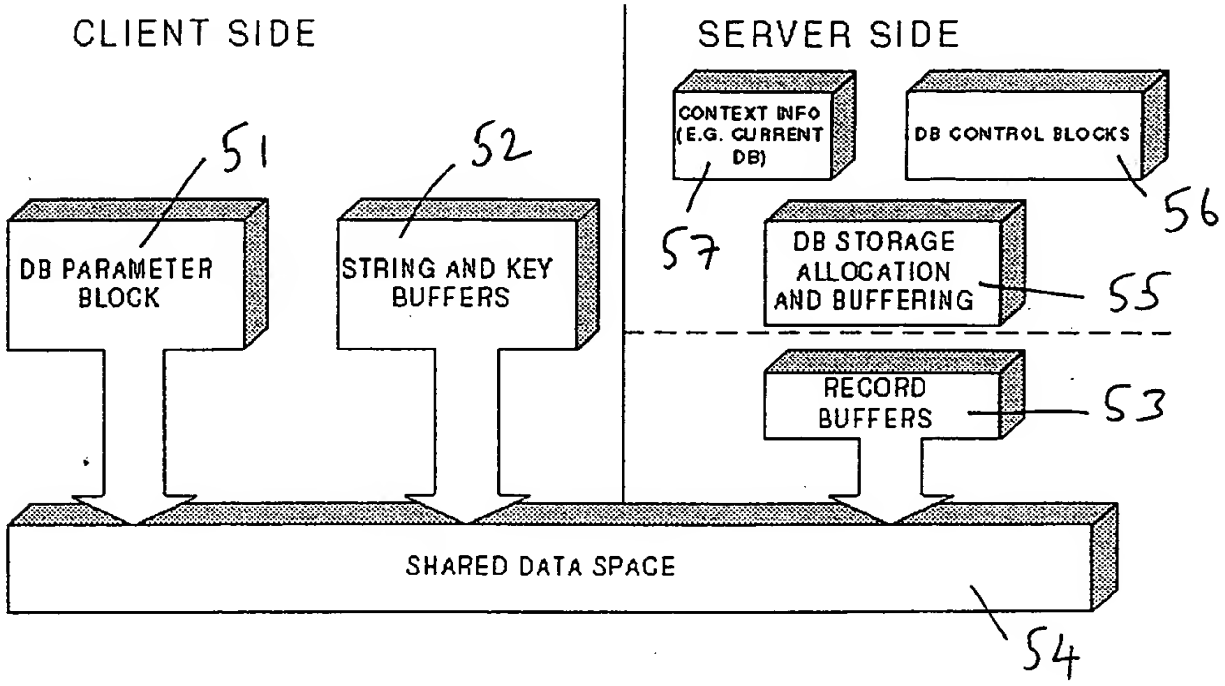

CLIENT SIDE
SERVER SIDE
CONTEXT INFO (E.G. CURRENT DB)
DB CONTROL BLOCKS
51
52
56
57
DB PARAMETER BLOCK
STRING AND KEY BUFFERS
DB STORAGE ALLOCATION AND BUFFERING
55
RECORD BUFFERS
53
SHARED DATA SPACE
54

Fig. 7

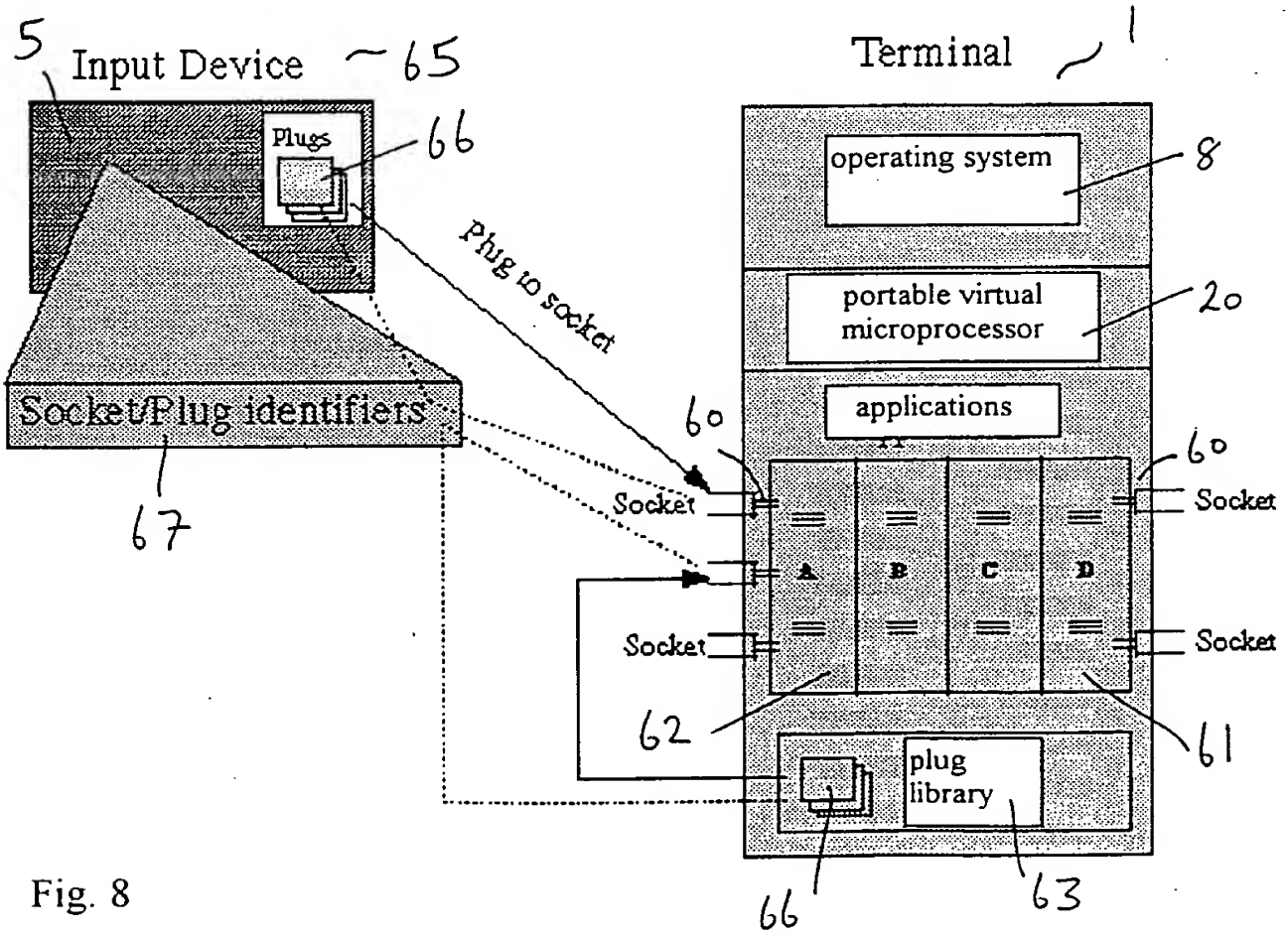

5
Input Device
65
Terminal
1
Plugs
66
operating system
8
Plug to socket
portable virtual microprocessor
20
Socket/Plug identifiers
applications
60
60
Socket
Socket
A
B
C
D
Socket
Socket
62
61
plug library
67
66
63

Fig. 8

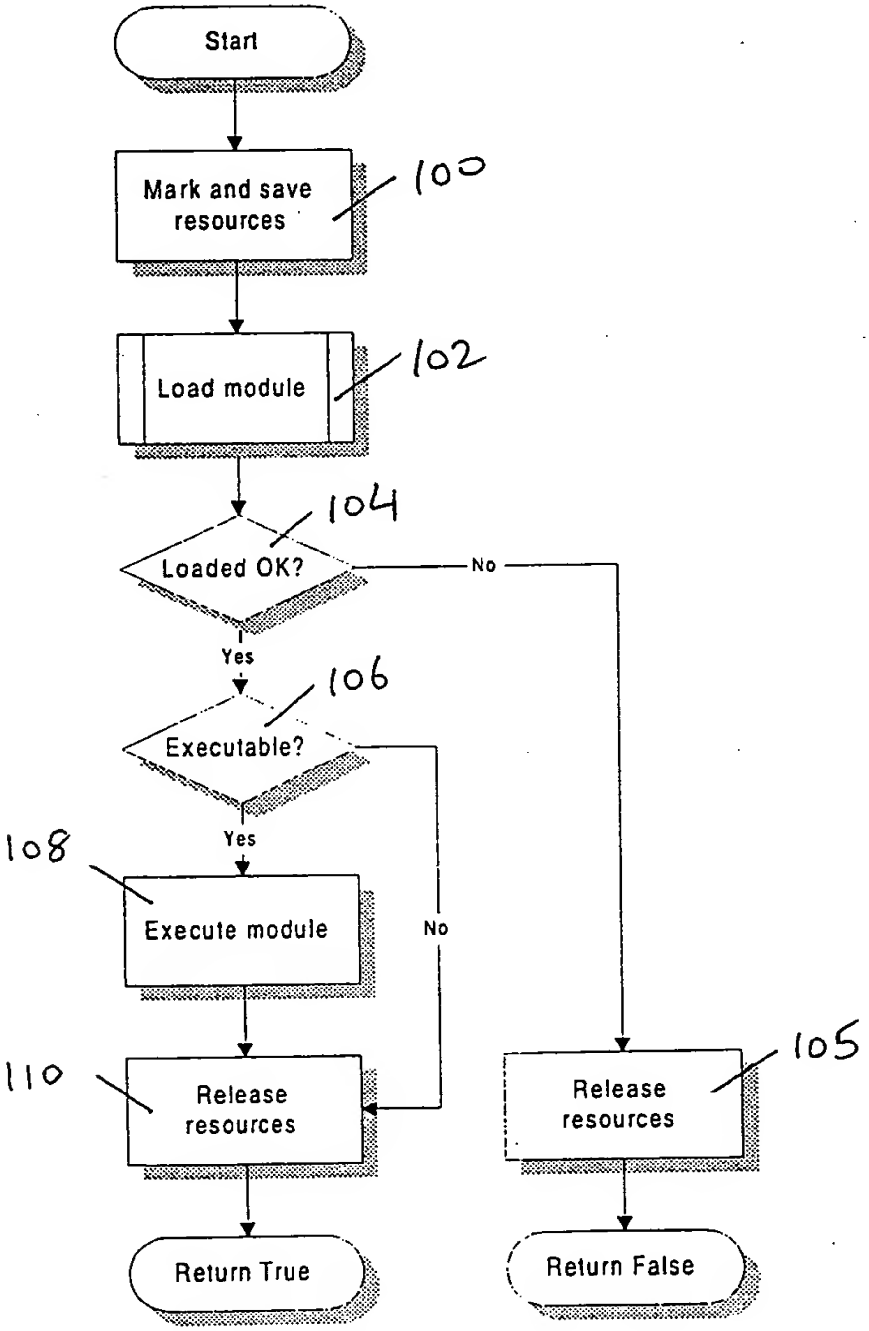

Start
Mark and save resources
100
Load module
102
104
Loaded OK?
No
Yes
106
Executable?
Yes
108
Execute module
No
110
Release resources
Return True
Release resources
105
Return False

FIG.9

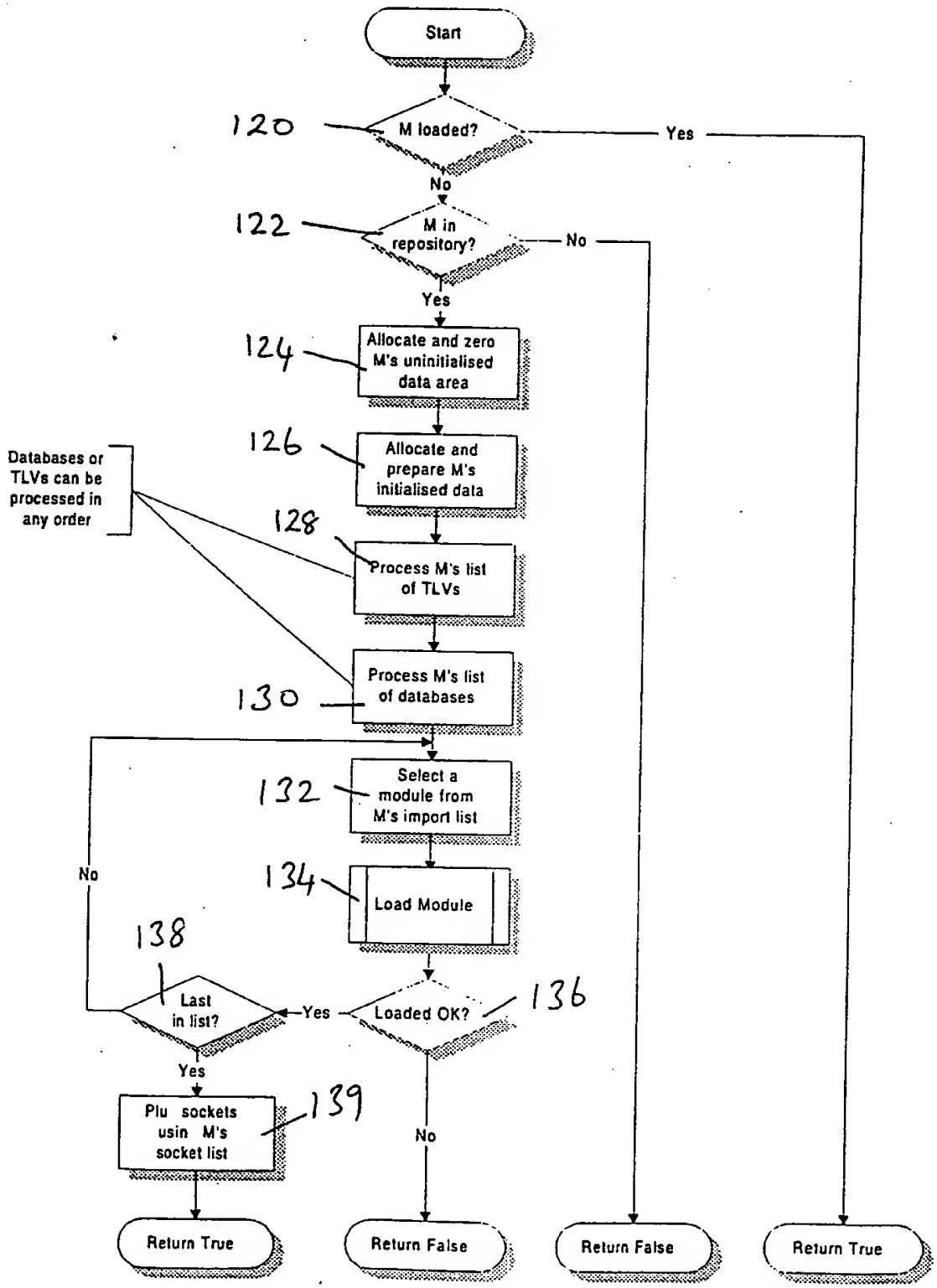

Start
120
M loaded?
Yes
No
122
M in repository?
No
Yes
124
Allocate and zero M's uninitialised data area
126
Allocate and prepare M's initialised data
Databases or TLVs can be processed in any order
128
Process M's list of TLVs
Process M's list of databases
130
132
Select a module from M's import list
No
134
Load Module
138
Last in list?
Yes
Loaded OK?
136
Yes
Plu sockets usin M's socket list
139
No
Return True
Return False
Return False
Return True

FIG. 10

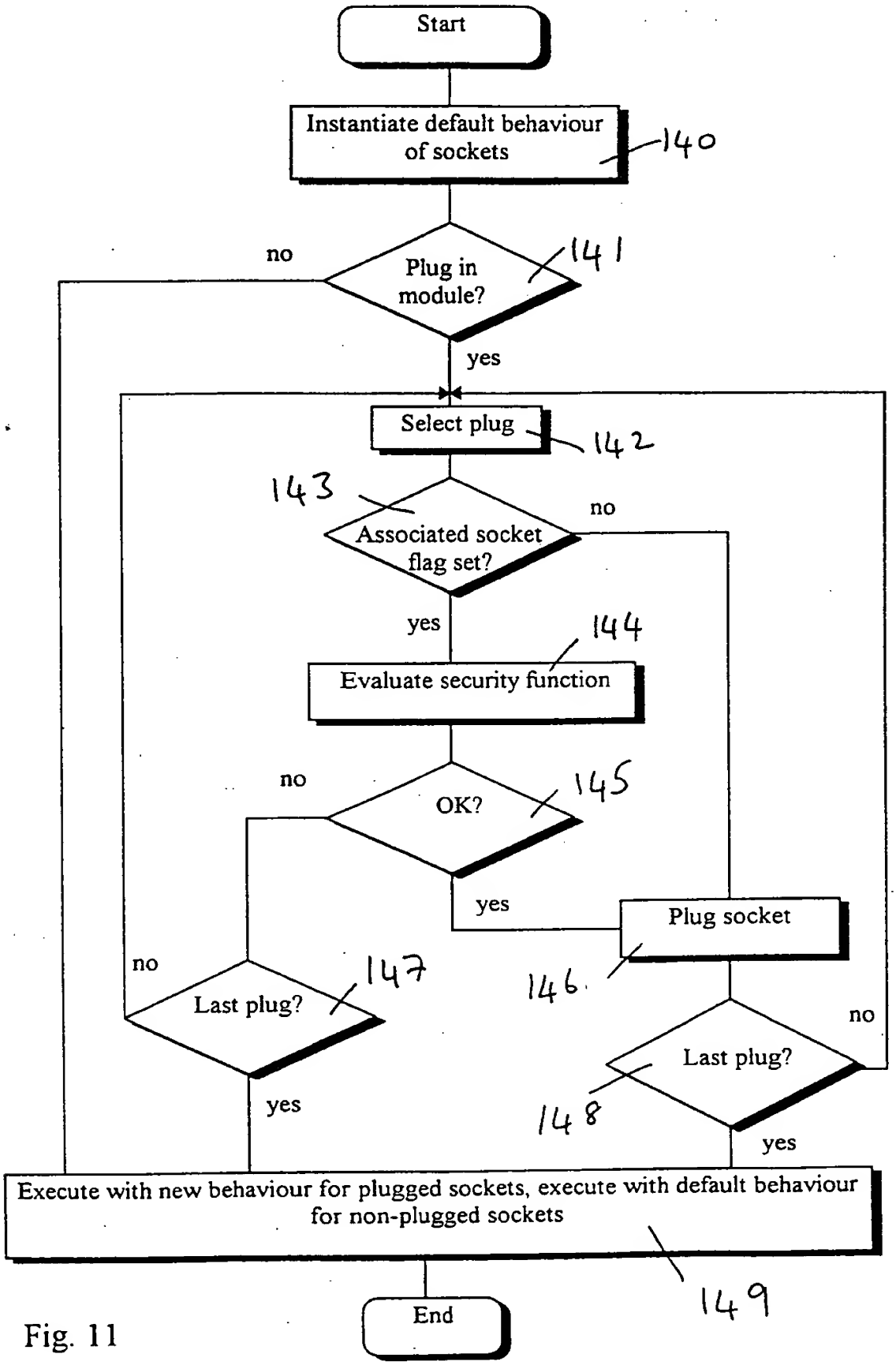

Start
Instantiate default behaviour of sockets
140
Plug in module?
141
no
yes
Select plug
142
143
Associated socket flag set?
no
yes
144
Evaluate security function
OK?
145
no
yes
Plug socket
146
Last plug?
147
no
yes
Last plug?
148
no
yes
Execute with new behaviour for plugged sockets, execute with default behaviour for non-plugged sockets
149
End

Fig. 11

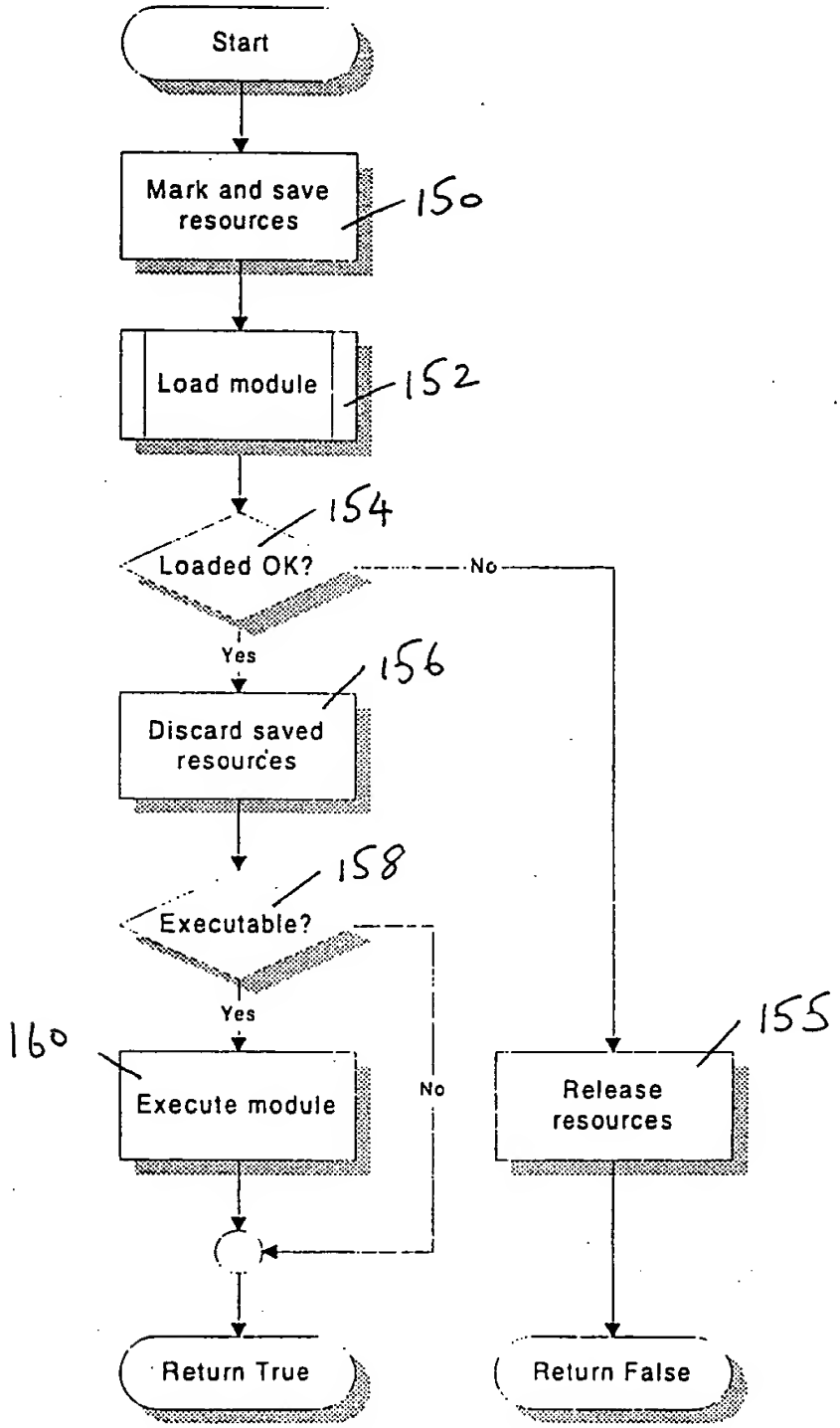

Start
Mark and save resources
150
Load module
152
154
Loaded OK?
No
Yes
156
Discard saved resources
158
Executable?
Yes
160
Execute module
No
155
Release resources
Return True
Return False

FIG. 12

PCT

WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : G07F 7/10, G06K 19/07	A3	(11) International Publication Number: **WO 97/50063** (43) International Publication Date: 31 December 1997 (31.12.97)

(21) International Application Number: PCT/EP97/03355

(22) International Filing Date: 26 June 1997 (26.06.97)

(30) Priority Data:
9613450.7 27 June 1996 (27.06.96) GB

(71) Applicant *(for all designated States except US)*: EUROPAY INTERNATIONAL N.V. [BE/BE]; Chaussée de Tervuren 198A, B-1410 Waterloo (BE).

(72) Inventors; and
(75) Inventors/Applicants *(for US only)*: HEYNS, Guido [BE/BE]; Eekhoornlaan 23B, B-3210 Linden (BE). JOHANNES, Peter [BE/BE]; Edelzangerslaan 56/11, B-3010 Kessel-Lo (BE).

(74) Agents: BIRD, William, E. et al.; Bird Goen & Co., Termerestraat 1, B-3020 Winksele (BE).

(81) Designated States: AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CU, CZ, DE, DK, EE, ES, FI, GB, GE, GH, HU, IL, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, TJ, TM, TR, TT, UA, UG, US, UZ, VN, YU, ZW, ARIPO patent (GH, KE, LS, MW, SD, SZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, ML, MR, NE, SN, TD, TG).

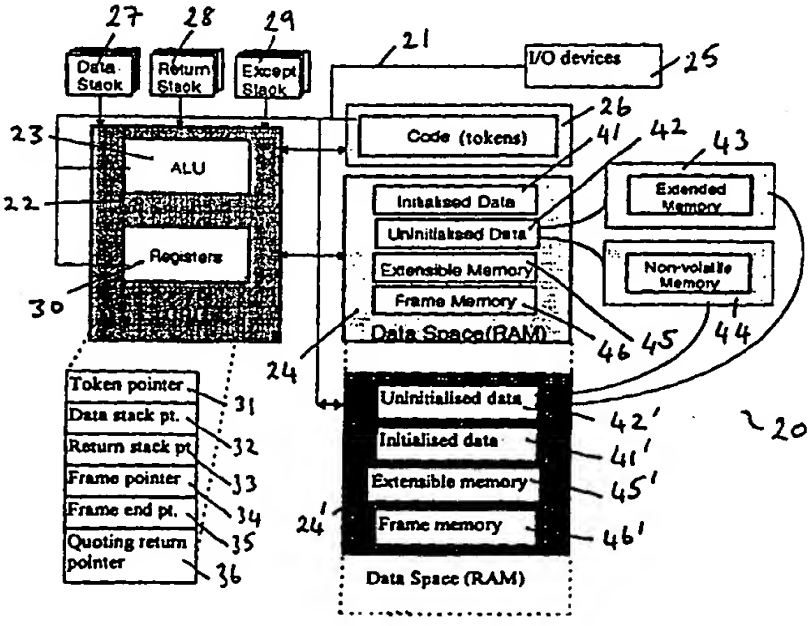
Published
With international search report.
Before the expiration of the time limit for amending the claims and to be republished in the event of the receipt of amendments.

(88) Date of publication of the international search report:
26 March 1998 (26.03.98)

(54) Title: PORTABLE, SECURE TRANSACTION SYSTEM FOR PROGRAMMABLE, INTELLIGENT DEVICES

(57) Abstract

The present invention provides a transaction management system for executing transactions between a first device (1) and a second device, said first and second devices being adapted for communication with each other and at least one of said first and second devices being an integrated circuit card, said system comprising: at least one input/output device (25); a portable virtual machine (20) for interpreting a computer program on said first device, said virtual machine comprising a virtual microprocessor and a driver for said at least one input/output device (25); and execution means responsive to said interpreted program for executing said program. The general linking technical concept behind the present invention is portability combined with security of data and run-time guarantees in a transaction system which are independent of the target implementation provided compile time checks are passed successfully. This concept is achieved by: using a virtual machine as an interpreter, including a driver for the I/O devices in the virtual machine so that application programs have a common interface with I/O devices and are therefore portable across widely differing environments, allocating and deallocating memory and including an indication of the amount of memory in the application program which means that the program will only run successfully or it will not run at all and security management functions are reduced to a minimum which improves operating speed, and providing a secure way of importing and exporting data in and out of application programs and databases.



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania
AM	Armenia
AT	Austria
AU	Australia
AZ	Azerbaijan
BA	Bosnia and Herzegovina
BB	Barbados
BE	Belgium
BF	Burkina Faso
BG	Bulgaria
BJ	Benin
BR	Brazil
BY	Belarus
CA	Canada
CF	Central African Republic
CG	Congo
CH	Switzerland
CI	Côte d'Ivoire
CM	Cameroon
CN	China
CU	Cuba
CZ	Czech Republic
DE	Germany
DK	Denmark
EE	Estonia
ES	Spain
FI	Finland
FR	France
GA	Gabon
GB	United Kingdom
GE	Georgia
GH	Ghana
GN	Guinea
GR	Greece
HU	Hungary
IE	Ireland
IL	Israel
IS	Iceland
IT	Italy
JP	Japan
KE	Kenya
KG	Kyrgyzstan
KP	Democratic People's Republic of Korea
KR	Republic of Korea
KZ	Kazakstan
LC	Saint Lucia
LI	Liechtenstein
LK	Sri Lanka
LR	Liberia
LS	Lesotho
LT	Lithuania
LU	Luxembourg
LV	Latvia
MC	Monaço
MD	Republic of Moldova
MG	Madagascar
MK	The former Yugoslav Republic of Macedonia
ML	Mali
MN	Mongolia
MR	Mauritania
MW	Malawi
MX	Mexico
NE	Niger
NL	Netherlands
NO	Norway
NZ	New Zealand
PL	Poland
PT	Portugal
RO	Romania
RU	Russian Federation
SD	Sudan
SE	Sweden
SG	Singapore
SI	Slovenia
SK	Slovakia
SN	Senegal
SZ	Swaziland
TD	Chad
TG	Togo
TJ	Tajikistan
TM	Turkmenistan
TR	Turkey
TT	Trinidad and Tobago
UA	Ukraine
UG	Uganda
US	United States of America
UZ	Uzbekistan
VN	Viet Nam
YU	Yugoslavia
ZW	Zimbabwe

INTERNATIONAL SEARCH REPORT

Interi nal Application No
PCT/EP 97/03355

A. CLASSIFICATION OF SUBJECT MATTER
IPC 6 G07F7/10 G06K19/07

According to International Patent Classification(IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 6 G07F G06K

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	US 5 434 999 A (C. GOIRE) 18 July 1995 cited in the application see the whole document ---	1,2,4,6, 16,17, 30-32, 44,45,57
A	EP 0 510 616 A (HITACHI) 28 October 1992 see abstract; claims; figures 1-3,46-50 see column 42, line 54 - column 44, line 15 ---	1-7,9, 44-46, 52-57, 61,64
A	WO 94 10657 A (INTELLECT AUSTRALIA) 11 May 1994 --- -/--	

[X] Further documents are listed in the continuation of box C.

[X] Patent family members are listed in annex.

° Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
28 January 1998	04/02/1998

Name and mailing address of the ISA	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016	David, J

Form PCT/ISA/210 (second sheet) (July 1992)

INTERNATIONAL SEARCH REPORT

Interl nal Application No
PCT/EP 97/03355

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category '	Citation of document. with indication,where appropriate, of the relevant passages	Relevant to claim No.
A	FR 2 667 171 A (GEMPLUS CARD INTERNATIONAL) 27 March 1992 ---	
A	US 5 067 072 A (K.K. TALATI) 19 November 1991 ---	
A	US 5 036 461 A (J.C. ELLIOTT) 30 July 1991 -----	

Form PCT/ISA/210 (continuation of second sheet) (July 1992)

INTERNATIONAL SEARCH REPORT

Information on patent family members

Inter nal Application No
PCT/EP 97/03355

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 5434999 A	18-07-95	FR 2638868 A	11-05-90
		CA 2002349 A,C	09-05-90
		DK 165390 A	22-08-90
		EP 0368752 A	16-05-90
		WO 9005347 A	17-05-90
		JP 7048178 B	24-05-95
		JP 3500827 T	21-02-91
		NO 300438 B	26-05-97
EP 0510616 A	28-10-92	JP 4322329 A	12-11-92
		US 5586323 A	17-12-96
WO 9410657 A	11-05-94	AU 5332194 A	24-05-94
		CA 2147824 A	11-05-94
		EP 0706692 A	17-04-96
		NO 951575 A	26-06-95
		US 5682027 A	28-10-97
FR 2667171 A	27-03-92	NONE	
US 5067072 A	19-11-91	US 4961133 A	02-10-90
		CA 1312959 A	19-01-93
		EP 0315493 A	10-05-89
US 5036461 A	30-07-91	NONE	